UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Model driven root cause analysis and reliability enhancement for large distributed computing systems

Michał Zasadziński

DOCTORAL THESIS

Model Driven Root Cause Analysis and Reliability Enhancement for Large Distributed Computing Systems

M.Eng. Michał Zasadziński

Computer Architecture Department

Universitat Politècnica de Catalunya

Barcelona, Spain

Supervisors:

Dr. David Carrera

Dr. Victor Muntés-Mulero

Thesis presented to obtain the qualification of Doctor from the Universitat Politècnica de Catalunya.

October, 2018, Barcelona

ACKNOWLEDGMENTS

I would like to thank many extraordinary and fantastic people for their input to the work of this thesis. Firstly, thanks to my wife, parents and the rest of my family. It could have been impossible to accomplish this mission without their support, understanding and fresh views. I would like to thank Victor Muntés-Mulero and Marc Solé for the mentorship and guidance they gave me. I could have observed what is called the right mentorship: "mentor stretches you, pushes for the risk, but at the same time shows the excellence and when you fail - support". It has been a pleasure to work with you. This PhD was intense and demanding, it had various phases, but the last one is exactly what I wish all young scholars and engineers: bravery and curiosity. Explore the World!

Michał

ABSTRACT

Over last years the number of Big Data, supercomputing, Internet of Things or edge systems has snowballed. The core part of many areas and services in academia and business, are large, distributed, and complex Information Technology (IT) systems. Any failure or performance degradation occurring in these systems causes negative effects. The user experience is decreased, operational costs raises, and a system loses its availability and readiness for demanding service. Also, there is a higher environmental footprint, *i.e.*, energy consumption. As the response, IT operators take care of resolving failures, issues, and unexpected events. Operators are aided with IT tools for monitoring, diagnostics, and root cause analysis. Usually, they have to troubleshoot and diagnose a system. Then, they perform some action to recover a system to its normal state.

However, the characteristics of the emerging and future IT systems makes the diagnostics and root cause analysis hard and complicated. These systems can contain even billions of elements, distributed all over the Earth. Moreover, systems can frequently change, consequently making existing diagnostic models outdated for the effective operation. Even the most skillful operators have problems to deal with these systems to satisfy QoS constraints and deliver spectacular user experience.

In this thesis, we contribute by making a step towards NoOps operating model. NoOps stands for no operations. In such a model of maintenance, an IT infrastructure is self-manageable and runs without human intervention. We would like to aid operator work and in the long term substitute them in the root cause analysis. We contribute for environments as mentioned earlier in two areas: (1) diagnostics, root cause analysis, root cause classification, and (2) prevention of failures. In particular, we focus on areas such as scalability, dynamism, lack of knowledge on system failures, predictability, and prevention of failures. For each of these aspects, we use different IT environment, for a proper diversification of the use cases.

Firstly, we propose a fast root cause analysis (RCA) system based on probabilistic reasoning. The system can diagnose networks of devices with millions of nodes in a diagnostic model and solves the problem of scalability of root cause analysis. In details, we leverage the fact that these systems usually contain repeatable elements. We create diagnostic models based on Bayesian networks. Then, we transform them into a more efficient structure for runtime use that are Arithmetic Circuits. Thanks to the proposed optimization in this transformation and cache-based mechanism, the solution performs better than state of the art techniques in terms of the memory consumption and time performance. Based on this contribution, we propose a root cause analysis system which works with dynamically changing environments. It is very common in Internet of Things environments, where devices connect and disconnect simultaneously, so the diagnostic model of the whole system changes rapidly. We propose actor based root cause analysis. This method is based on distributing diagnostic calculations through the devices and use of self-diagnostics paradigm. Thanks to this solution, results of partial diagnosis are known even when the connectivity with a part of the diagnosed system is lost. Also, we provide a framework to define supervising strategies which are utilized in case of a failure. We show that the contribution works well in a simulated Internet of Things system with high dynamism in its structure.

Secondly, we focus on the aspect of knowledge integration and partial knowledge of a diagnosed system. The path to NoOps involves not only precise, reliable and fast diagnostics but also reusing as much knowledge as possible after the system is reconfigured or changed. The biggest challenge is to leverage knowledge on one IT system and reuse this knowledge for diagnostics of another, different system. We propose a weighted graph framework which can transfer knowledge and perform high-quality diagnostics of IT systems. We encode all possible data in a graph representation of a system state and automatically calculate weights of these graphs. Then, thanks to the similarity evaluation, we transfer knowledge about failures from one system to another and use it for diagnostics. We successfully evaluate the proposed approach on Spark, Hadoop, Kafka and Cassandra systems. For this purpose, we use an on-premise Big Data cluster and a cloud system of containers.

Thirdly, we focus on the predictability of a supercomputing environment and prevention of failures. Failed jobs in a supercomputer cause not only waste in CPU time or energy consumption but also decrease work efficiency of users. Mining data collected during the operation of data centers helps to find patterns explaining failures and can be used to predict them. Automating system reactions, *e.g.*, early termination of jobs, when software failures are predicted does not only increase availability and reduce operating cost, but it also frees administrators' and users' time. We explore a unique dataset containing the topology, operation metrics, and job scheduler history from the petascale Mistral supercomputer. We extract the most relevant system features deciding on the final state of a job through decision trees. Then, we successfully propose actions to prevent failures. We create a model to predict job evolution based on power time series of nodes. Finally, we evaluate the effect on CPU time saving for static and dynamic job termination policies. We finish the thesis with a short discussion on the contributions and state directions for future work.

RESUMEN

En los últimos años, la cantidad de Big Data, supercomputación, dispositivos IoT o sistemas *edge* se ha disparado. Grandes sistemas distribuidos y complejos de tecnología de la información (TI) forman la parte central de muchas áreas y servicios en el mundo académico y la industria. Cualquier falla o degradación del rendimiento que ocurra en estos sistemas puede acarrear importantes efectos adversos: una experiencia de usuario empobrecida, costos operativos que aumentan y pérdida de disponibilidad del sistema y su capacidad para operar en condiciones óptimas. Además, los fallos pueden contribuir a una huella ambiental más alta, a través de un mayor consumo de energía. Como respuesta, los operadores de TI se encargan de resolver fallas, problemas y eventos inesperados. Se ayuda a los operadores con herramientas de TI para el monitoreo, el diagnóstico y el análisis de causa raíz. Por lo general, primero tienen que solucionar y diagnosticar un sistema, y luego realizan alguna acción para recuperar el sistema a su estado normal.

Sin embargo, las características de los sistemas de TI emergentes y futuros dificultan y complican el diagnóstico y el análisis de la causa raíz. Estos sistemas pueden contener incluso miles de millones de elementos distribuidos por toda la Tierra. Además, los sistemas pueden cambiar con frecuencia, lo que hace que los modelos de diagnóstico existentes estén obsoletos mermando así su efectividad. Incluso los operadores más hábiles tienen problemas para lidiar con estos sistemas para satisfacer los niveles de calidad de servicio (QoS) esperados y ofrecer una experiencia de usuario impecable.

En esta tesis, contribuimos dando un paso hacia el modelo operativo NoOps. NoOps no significa no realizar ninguna operación, sinó que una infraestructura de TI sea autogestionable y funcione sin intervención humana. Nos gustaría ayudar al trabajo del operador y, a largo plazo, sustituirlo por un sistema automatizado de análisis de la causa raíz. En ese sentido contribuimos en dos áreas: (1) el diagnóstico, análisis de causa raíz basado en clasificación, y (2) prevención de fallas. En particular, nos enfocamos en áreas tales como escalabilidad, dinamismo, falta de conocimiento sobre fallas del sistema, previsibilidad y prevención de fallas. Para cada uno de estos aspectos, utilizamos un entorno de TI diferente para diversificar los casos de uso.

En primer lugar, proponemos un sistema rápido de análisis de causa raíz (RCA) basado en razonamiento probabilístico. El sistema puede diagnosticar redes de dispositivos con millones de nodos en un modelo de diagnóstico y resuelve el problema de la escalabilidad del análisis de causa raíz. En particular, aprovechamos el hecho de que estos sistemas generalmente contienen elementos repetidos. Creamos modelos de diagnóstico basados en redes Bayesianas para estos elementos repetidos. Luego, los transformamos en una estructura más eficiente en tiempo de ejecución que son los Circuitos Aritméticos. Gracias a la optimización propuesta en este mecanismo de transformación y basado en caché, la solución funciona mejor que las técnicas más avanzadas en términos de consumo de memoria y tiempo. Con base en esta contribución, proponemos un sistema de análisis de causa raíz que funciona con entornos que cambian dinámicamente. Esto es muy común en los entornos de IoT, donde los dispositivos se conectan y desconectan con frecuencia, por lo que el modelo de diagnóstico de todo el sistema cambia rápidamente. En este entorno proponemos un análisis de causa raíz basado en actores. Este método se basa en la distribución de cálculos de diagnóstico a través de los dispositivos y el uso del paradigma de autodiagnóstico. Gracias a esta solución, los resultados del diagnóstico parcial se conocen incluso cuando se pierde la conectividad con una parte del sistema diagnosticado. Además, brindamos un marco para definir estrategias de supervisión que se utilizan en caso de falla. Mostramos que la contribución funciona bien en un sistema IoT simulado con un alto dinamismo en su estructura.

En segundo lugar, nos centramos en el aspecto de la integración del conocimiento y el conocimiento parcial de un sistema diagnosticado. El camino hacia NoOps implica no solo diagnósticos precisos, confiables y rápidos, sino también la reutilización del mayor conocimiento posible después de que el sistema se reconfigure o cambie. El mayor desafío es aprovechar el conocimiento en un sistema de TI y reutilizar este conocimiento para el diagnóstico de otro sistema diferente. Proponemos una aproximación basada en grafos con pesos que puede transferir conocimiento entre sistemas diferentes y realizar diagnósticos de alta calidad de los sistemas de TI. Codificamos todos los datos posibles en un grafo de un estado del sistema y calculamos automáticamente los pesos de asociados a los elementos de este grafo. Luego, gracias a una función de similitud entre grafos, transferimos el conocimiento sobre fallas de un sistema a otro y lo usamos para el diagnóstico. Evaluamos con éxito el enfoque propuesto en los sistemas Spark, Hadoop, Kafka y Cassandra. Para este propósito, usamos un clúster para Big Data (con acceso físico) y un sistema de contenedores en la nube.

En tercer lugar, nos centramos en la previsibilidad de un entorno de supercomputación y la prevención de fallas. Los trabajos fallidos en una supercomputadora no solo causan pérdidas en el tiempo de CPU o en el consumo de energía, sino que también disminuyen la eficiencia del trabajo de los usuarios. Los datos recopilados durante la operación de los centros de datos pueden ayudar a encontrar patrones que expliquen fallas y pueden usarse para predecirlos. La automatización de las reacciones del sistema, por ejemplo, la terminación anticipada de trabajos, cuando se prevén fallas de software, no solo aumenta la disponibilidad y reduce los costos operativos, sino que también libera el tiempo de los administradores y los usuarios. Exploramos un conjunto de datos único que contiene la topología, las métricas de operación y el historial del planificador de tareas del superordenador Mistral. Extraemos las características más relevantes del sistema para decidir sobre el estado final de un trabajo a través de árboles de decisión. Luego, proponemos acciones para evitar fallas. Creamos un modelo para predecir la evolución del trabajo basado en la serie temporal de potencia de los nodos. Finalmente, evaluamos el efecto sobre el ahorro de tiempo de la CPU para las políticas de terminación de trabajos estática y dinámica. Terminamos la tesis con una breve discusión sobre las contribuciones de esta tesis y consideraciones de posible trabajo futuro.

PREFACE

Last decades, global computerization directed all areas of our life. Complicated, large, distributed, and heterogeneous computing systems, *e.g.*, supercomputers, Big Data, Internet of Things, Cyber-Physical Systems serve human in diverse domains. Even, people who are not engaged in IT research or industry are exposed to the IT world. For instance, as simple citizens, we use *e.g.*, smart transportation, smart health-care systems, and crowd-sourced systems. However, no matter what the structure and scale of these systems is, all of them have two things in common. First, they all face failures degrading their performance and reliability. Second, human need to take care of diagnostics, reconfiguration, and improvement of these systems. Finally, in a considerable part, the quality of a service depends on the reaction and skills of operators. However, as the growth of IT systems is theoretically unlimited, the capacity of the human brain is. We have already reached the point when human is not able to maintain overgrown systems. Is it possible that these systems take care of themselves automatically without the people engagement?

"One never notices what has been done; one can only see what remains to be done."

— Marie Skłodowska-Curie

"The voyage of discovery is not in seeking new landscapes but in having new eyes."

— Marcel Proust

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

ABRCA — Actor Based Root Cause Analysis

AC — Arithmetic Circuit

AI — Artificial Intelligence

BN — Bayesian Network

CPT — Conditional Probability Table

CNN — Convolutional Neural Network

CPS — Cyber Physical System

DT — Decision Tree

DL — Deep Learning

DAG — Directed Acyclic Graph

IoT — Internet of Things

JVM — Java Virtual Machine

ML — Machine Learning

MPE — Most Probable Explanation

NN — Neural Network

PRM — Probabilistic Relational Models

RCA — Root Cause Analysis

TCO — Total Cost of Ownership

CHAPTER 1

INTRODUCTION

Today, Information Technology (IT) systems have already become a fundamental part of many activities and services in business, academia and daily life. These systems have different scales, structures and types of infrastructure. However, it does not prevent us from saying that reliability is a key and common aspect of running all of them. For instance, let us think about Internet of Things (IoT) devices communicating with cloud services running on centralized data centers. This system is complex and dynamic, many failures cannot be resolved in seconds or minutes, and the user demand for a service is not satisfied. Another example are business applications, whether on-premise or in the cloud, failures usually causes expensive business service delays. Also, consider supercomputers, where failures of jobs decrease resource availability and increase operational costs. Any failure or performance degradation occurring in these systems causes negative effects.

Because of failures and low reliability of a system, the user demand is not satisfied, operational costs are increased, and a system loses its availability and readiness for the service. Also, there is a higher environmental footprint, *i.e.*, energy consumption. As a response, IT operators take care of resolving failures, issues, and unexpected events. Operators are aided with different IT tools used for monitoring, diagnostics, and root cause analysis. Then, they perform some action to recover a system back to its normal operation. However, some systems are not in a reach of troubleshooting capabilities of even the most skilled and experienced operators. Operators have problems to deal with these systems to satisfy QoS constraints and deliver acceptable user experience.

In this thesis, we contribute by making a step towards NoOps operating model. NoOps stands for no operations. NoOps is focused on externalizing the operation of the system and then removing operations from the infrastructure at the user side. One of the ways to implement such a model of maintenance is to run a self-manageable IT infrastructure and without human intervention. We would like to aid operator work and in the long term substitute them in the root cause analysis. We contribute for environments as mentioned earlier in two areas: (1) diagnostics, root cause analysis, root cause classification, and (2) prevention of failures. In particular, we focus on areas such as **scalability, dynamism, lack of knowledge on system failures, predictability, and prevention of failures**. For each of these aspects, we use different IT environment, proving that our solutions are transversal across multiple use cases. In particular, we research with a petascale supercomputer, Big Data cluster, containers, and a simulated Internet of Things environment.

1.1 A step towards NoOps: Model-driven automation and reliability enhancement for large distributed computing systems

Reacting to failures and recovering a system fast is the main objective of every IT system operator. Many IT tools support operators to troubleshoot and resolve issues in IT systems. However, to step further and transform the characteristics of operators' work, the problems occurring while working with complex and large environments should be resolved. Depending on the type of a system, an operator and troubleshooting tools might face some of the following problems.

- **scalability** – system scales up significantly. How should a diagnostics framework, including a system model and diagnosing algorithm scale?
- **dynamism** – a system changes its structure and configuration rapidly and in an unpredictable way. How to diagnose such a system to provide an updated diagnosis?
- **partial or unknown diagnostic model of a system** – a system has been already set up, or it has been reconfigured. How to diagnose it from its start, without having neither observations on failures nor the diagnostic model?
- **predictability** – is it possible to predict failures? How early is it possible to prevent them?

Addressing the above issues cannot only increase the effectiveness of diagnostics and system recovery, but also it can help in automatizing it. Moreover, a human can be eliminated from a diagnostic process which is often repetitive but complex. In this thesis, we do a step on the way to NoOps. It means an operational model of an IT system where no human is engaged in maintenance. All maintenance processes are performed automatically, and an IT system is self-manageable.

In Figure 1.1 we show a graph presenting a holistic view of problems of reliability of an IT system and contributions presented in this thesis. IT systems are often under continuous development and change, in order to satisfy a user's demand and requirements. These processes may result in reconfiguration of a system, scaling and failures, *e.g.*, caused by software errors. The resulting problems such as occurring failures, difficult and resource-consuming troubleshooting are responsible for even worse consequences. The most significant ones are decreased QoS and resource availability, increased Total Cost of Ownership (TCO), energy consumption and environmental footprint. Directly, these consequences can result in, *e.g.*, the necessity to engage more workforce for troubleshooting, repeat failed jobs, recover a system from backups.

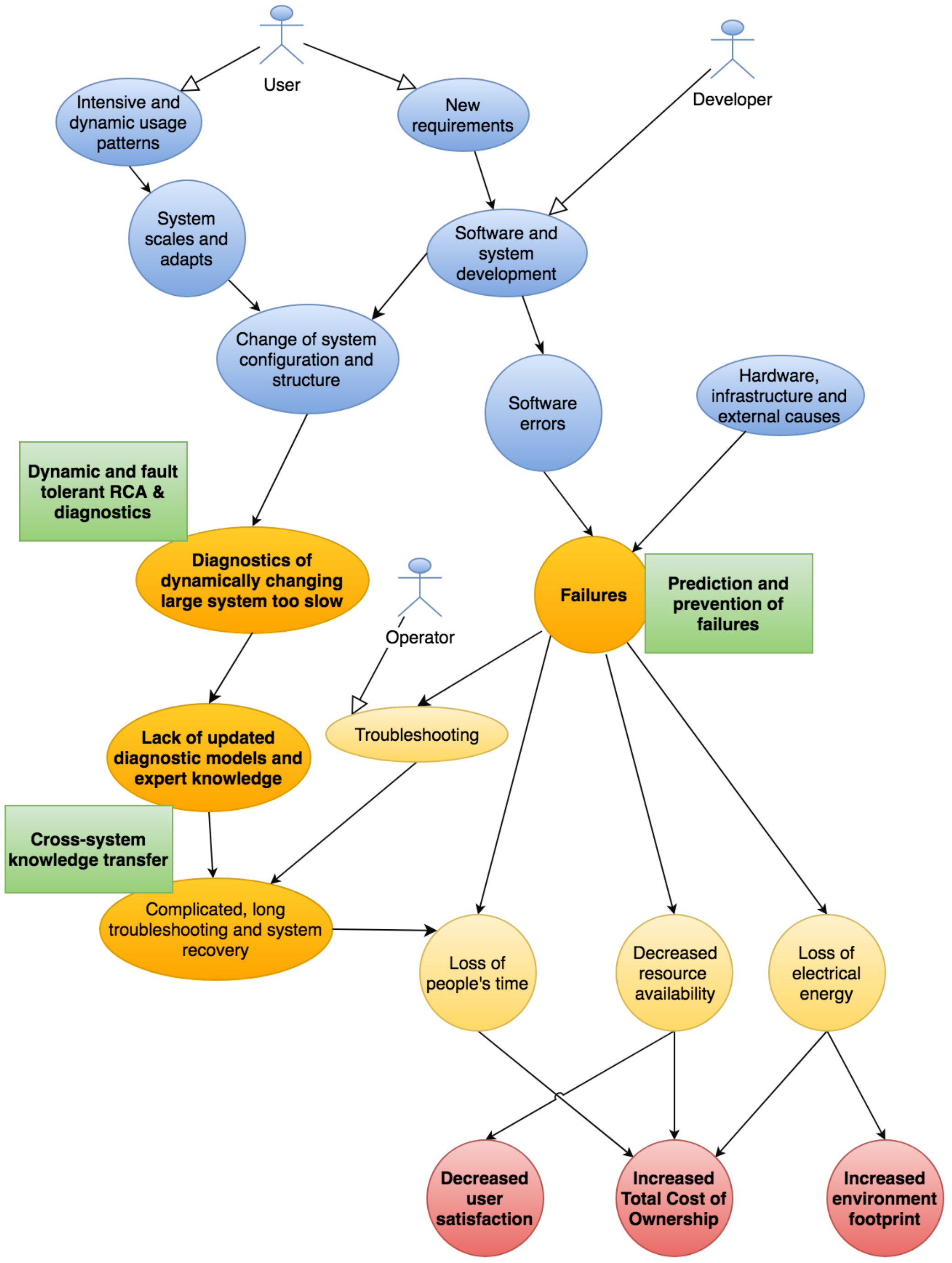

User
Developer
Intensive and dynamic usage patterns
New requirements
System scales and adapts
Software and system development
Change of system configuration and structure
Software errors
Hardware, infrastructure and external causes
Dynamic and fault tolerant RCA & diagnostics
Diagnostics of dynamically changing large system too slow
Operator
Failures
Prediction and prevention of failures
Troubleshooting
Lack of updated diagnostic models and expert knowledge
Cross-system knowledge transfer
Complicated, long troubleshooting and system recovery
Loss of people's time
Decreased resource availability
Loss of electrical energy
Decreased user satisfaction
Increased Total Cost of Ownership
Increased environment footprint

Figure 1.1: Graph presenting detailed causes (blue), problems (yellow and orange) and consequences (red) in IT systems. Green rectangles represent solutions presented in this Thesis.

1.2 Contributions

We take a challenge of aiding human in complex diagnostics. The objectives of the thesis are:

- Enable fast root cause analysis in large, distributed environments
- Enable diagnostics with missing observations and knowledge
- Automate system reaction for predicted failures

The contributions of this thesis are the following.

1. Root cause analysis system based on split Bayesian networks and their transformation that are Arithmetic Circuits. The system leverages the fact that in large IT systems there are many repeated components, it manipulates Arithmetic Circuits and caches the partial results and structures. The system scales to millions of nodes in a diagnostic model. (**Chapter 4**)

2. Actor based root cause analysis system. The system uses resources of the devices making up a system for root cause analysis calculations. Actor based model allows for the implementation of different policies in case of failures. (**Chapter 4**)

3. An integration of metrics, logs and static information to a weighted graph that represents the state of a diagnosed system. Logs are encoded automatically as node attributes without the necessity to define any metadata. (**Chapter 5**)

4. A framework for knowledge transfer using weighted graphs. The framework can find the closest repository graph that describes the runtime failure, even if a diagnosed system has different structure and functionality than a graph representing previously registered failure. (**Chapter 5**)

5. Insights and results of data mining of the operations of a petascale supercomputing environment. The analysis includes the discovery of trends, phenomena, and correlations leading to the explanation of a job final state. We discover importance of features and data sources deciding and predicting a job final state. (**Chapter 6**)

6. Static and dynamic job failure prevention policies. The static policy is created using Decision Trees, while the dynamic one uses a Convolutional Neural Network. The static policy can predict failed jobs using all information known at the time of a job submission, including historical data. The dynamic one uses only power metrics of nodes allocated to a job. (**Chapter 6**)

1.3 Contents of the document

The content of the document is the following. Firstly, we provide introduction to the research problem in **Chapter 1** and describe background of the research and related work in **Chapter 2**. Then, we describe important IT environments for which we target our solutions in **Chapter 3**. The environments include a supercomputer, Big Data cluster, system of containers in a cloud, and a simulated IoT system. We use these environments for experiments and evaluations in the next sections.

In **Chapter 4** we address the problem of the **scalability** of diagnosing large IoT environments. We propose a scalable and fault tolerant root cause analysis framework for distributed large heterogeneous environments. Firstly, we propose a solution based on probabilistic reasoning. Thanks to the proposed approximate modeling and reasoning, the proposed method can deal with diagnostic models consisting of millions of elements. Then, we propose a distributed root cause analysis framework which deals with **dynamically** changing IoT environments. The framework can perform root cause analysis of an IoT system with constantly connecting and disconnecting devices. We evaluate both approaches on a simulated IoT system.

In **Chapter 5** we address the problem of diagnosing a system when there is **partial knowledge** about symptoms of a failure. We contribute by root cause classification system which performs cross-system diagnostics through knowledge transfer. Firstly, we present a graph similarity framework. Then, using this framework we represent a system state integrating logs, metrics and system topology. We show how this framework is used for root cause classification through evaluation of the similarity between two graphs. One graph represents a state of a monitored system, the other one a system state during a failure. We evaluate this framework on a Big Data cluster running Spark [2] and Hadoop [3]. Then, we show how the proposed graph framework can be used for transferring knowledge from one system to another. Thanks to the proposed graph framework, we acquire knowledge from one system and diagnose completely another system. As an example, we evaluate knowledge transfer approach on a cloud environment running containers and micro-services, in particular, Kafka [4] and Cassandra [5].

In **Chapter 6** we explore predictability of a supercomputing environment and propose a framework for prevention of failures. Firstly, we explore how predictable a supercomputing environment is. Through advanced data mining, we detect trends and phenomena describing user-application-supercomputer ecosystem. Also, we show the most meaningful job features deciding on the final state of a job. Then, we contribute with a framework for prevention of failures. We propose a static and dynamic policy for early termination of jobs predicted for failure. For a static policy, we use Decision Trees, which allow creating a white-box explainable model of failed jobs. Thanks to this model, we can evaluate jobs at the time of their submission. For a dynamic one, we use a model based on Convolutional Neural Networks. Thanks to this model, we can predict failures during the runtime of a job, and take appropriate actions based on these predictions.

In **Chapter 7** we provide the wrap-up and propose directions of the future work. Also, at the end of each chapter, we provide detailed conclusions and particular future work plans. At the end of the thesis, we list dissemination activities performed within doctorate studies.

CHAPTER 2

RELATED WORK

In this chapter, we describe work related to the research field of this thesis. In Section 2.1, we focus on root cause analysis through networks of plausible inference and Probabilistic Relational Models [6]. We focus on state of the art of root cause analysis and diagnostics for a use case of the Internet of Things and Cyber-Physical systems. Then, in Section 2.2 we study work related to graph-based root cause analysis and root cause classification systems. We focus on a graph representation of a system state including metrics and logs. Also, we explore work related to cross-system knowledge transfer. In Section 2.3 we study work related to prediction and analysis of failures in HPC environment. In particular, we focus on different machine learning techniques. Also, we explore creating different models with the use of logs for diagnostics.

2.1 Root cause analysis of distributed systems: graph approaches for handling large and dynamic models

Methods like root cause analysis support operations of large and complex environments. In this Section, we describe existing solutions and research mainly focusing on the Internet of Things and Cyber-Physical Systems. Regarding the particular computational and modeling framework, we focus on Networks of Plausible Inference. We start with a study of related work on Bayesian networks, go through Probabilistic Relational Models, and we explore the most critical research on Bayesian networks which leads to large-scale deployment of these networks.

2.1.1 Root cause analysis for the Internet of Things and Cyber-Physical Systems

High-performance root cause analysis for large distributed environments is an active research topic. The authors of [7–9] focus on automatic analytics of events and black-box models of cloud environments. Rarely, publications present work which uses probabilistic relational models for troubleshooting large environments. However, as we show later on, probabilistic reasoning is a practical framework for root cause analysis. A conventional approach for implementing root cause analysis is using classification algorithms and algorithms based on the correlation of alarms [10]. Research presented in [11] provides a graph-based root cause analysis system, and the authors use it on a large distributed system of servers. For instance, the authors of [12] propose a root cause analysis system for complex enterprise networks. The presented research introduces the idea of constructing a causality graph between events in the system which is used for failure localization. The challenges of using root cause analysis systems are accuracy, diagnosis time, tractability and scalability of these solutions.

Also, there is a growing amount of research that considers root cause analysis in the Internet of Things ecosystem. However, performing such analysis considering all the tiers presented in Figure 3.1 is not intensely studied in the literature. Each of these layers has different components and topologies. However, a common characteristic of these systems is that they contain a lot of repeated elements. This fact should be utilized by diagnostic systems to improve their performance during the runtime. In [13] authors summarize the main research motivations for the reliability of cloud services. They state the difficulty of detection of failures and faults in the cloud; they mention the problem of little research on scalable fault detection methods, and difficulties in recognizing the causes of failures. Besides, Aggarwal [14] states the problem of having incomplete data transmitted from sensors and the significance of this problem for Big Data analytics. In [15] it is stated that failures in fog computing can be localized in sensors, network, lack of network coverage, a service platform or the web application. Authors in [16] propose the integration of Big Data with a Cyber-Physical System, describe a data-driven approach to building fault tolerant control systems and they emphasize its significance via research. Moreover, the accurate mathematical models, will not be able to deal with the scale and computational complexity of large Internet of Things structures.

Furthermore, many researchers focus on root cause analysis for Cyber-Physical Systems [17]. The authors of [17] describe Cyber-Physical Systems as systems that integrate computing, communication and storage capabilities with physical processes, monitoring, and control. CPSs should perform their activities securely, efficiently and in real-time. CPSs have to distribute computations while having effective network control [17]. The described systems are characterized by a high degree of integration of physical devices and large geographical dispersion. These issues imply that controlling and monitoring need to be highly distributed. For instance, these properties are observed in IoT and Smart Cities applications. The authors state that new tools and algorithms should be created for fault analysis in the mentioned environments. Additionally, authors of [18] emphasize that the reliability of CPSs is an active problem to address.

The idea of using resources of a system for diagnostics is widely described in [19]. The authors propose distributed diagnostics for wireless sensor networks. The considered solution is motivated by the loss of information transferred from sensors to the central module of the system. However, due to highly limited resources of sensors (tens of KB of memory), complex models cannot be used for diagnostics. Another approach of root cause analysis system for CPS is presented in [20]. The authors propose a system that performs hierarchical fault reasoning. The system starts with light diagnostic calculations and performs more complex ones when necessary. However, the solution does not consider dynamically changing large-scale systems. For instance, a proactive health monitoring system is presented in [21]. It is principally motivated by the necessity of a quick and dependable response during diagnostics and optimal resource consumption in limited working conditions. The authors propose an adaptive diagnosis quality driven solution of the sensor activation problem by choosing an optimal reconfiguration of the diagnosing system. Nevertheless, the researchers use a rule-based approach and a sparse matrix which can cause calculation problems in extreme scale systems, where the number of components is on the hundreds of thousands.

2.1.2 Fundamental networks of plausible inference: Bayesian networks for root cause analysis

One of the underlying frameworks for creating and inferring diagnostic models are Bayesian networks [6]. An example Bayesian network is presented in Figure 2.1. A BN helps to describe the probability of some events happening, given their probabilistic conditional dependencies on other events. For instance, the example BN, can solve top-down problems like ”What is the probability that the device will disconnect, once the battery status is low and network coverage is medium?” Also, it may solve bottom-up (root cause analysis) problems: ”Device has disconnected. Why? Which event is the cause? Which event was more likely to happen: the device is overheated, or edge dropped the connection?”

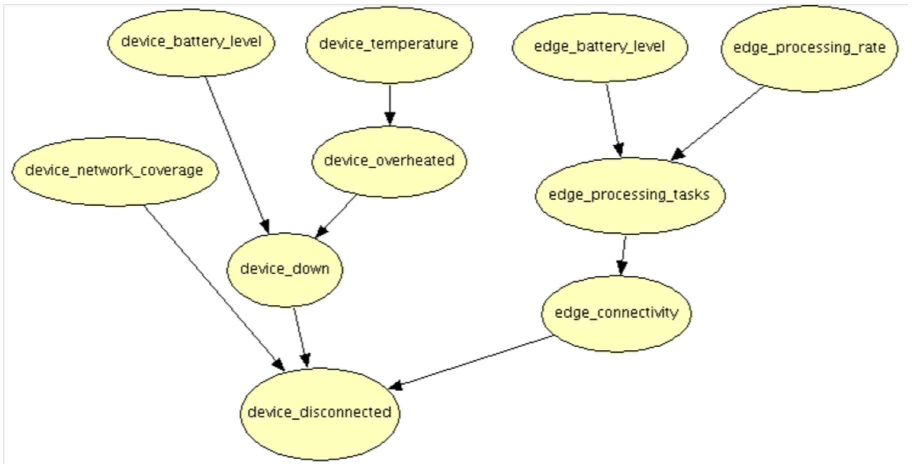


Figure 2.1: Example Bayesian network (BN) representing diagnostic model for a device connected to an edge device.

Formally, a Bayesian network is a probabilistic network which describes relations between probabilistic variables in the form of a Directed Acyclic Graph. Variables used in Bayesian networks can be categorical (finite number of values), *e.g.*, *error_type* having value ”kernel” or ”system”; or continuous, *e.g.*, temperature, CPU load. In the case of categorical variables, we define probabilities with tables. Each node contains a conditional probability table (CPT) that describes conditional probabilities of states of this node depending on the states of the parent nodes. Bayesian networks represent well dependencies between causes and symptoms. BNs are often used when it is hard to create an exact analytic or simplified model of a system. They are used widely in the modeling of events and phenomena in many different domains, *e.g.*, IT, medicine, finance, sport. Primarily, they are used where diagnostic and reasoning models should leverage expert knowledge, and enough data is given to build distributions of the variables or calculate their probabilities.

When compared to other diagnostic techniques [22], the Bayesian networks are distinguished as a solution for problems of an unacceptable quantity of false alarms. These alarms can be set off by a monitoring system with a threshold approach. One of the most significant advantages of using Bayesian networks is that a system with Bayesian reasoning can provide early alerts before the fault occurs. It is especially useful in systems where many faults do not develop gradually over time. Rather they occur instantaneously. In [23], authors describe software health monitoring system using the BNs designed for monitoring, diagnosis, and prediction in the software-hardware environment. The designed system meets the requirements of being powerful enough to reliably detect and localize significant failures with a provision of advanced reasoning, but the research does not include large-scale deployment.

Another essential publication [24] on BN presents large-scale deployment of a diagnostic system for web applications. The solution is based on Bayesian networks and noisy-OR nodes, and it uses approximate reasoning for fast diagnosis. Acceptable accuracy characterizes the solution. Also, a significant industrial deployment and research on inference through BN is presented in [25]. The authors optimize BN reasoning for large-scale Virtual Private Networks. They perform failure diagnostics, using reasoning in anomalous regions of a failed node.

2.1.3 Object relationship modeling with Bayesian networks: Probabilistic Relational Models

When used for large systems, Bayesian networks are even more complicated. This makes them difficult to create or update. The solution is modeling objects with underlying Bayesian networks. The idea of splitting BNs into objects that are related to the modeled components is well introduced and explained in the literature in the area of Probabilistic Relational Models (PRM) [26, 27]. In this framework, a reasonably significant amount of work on structured probabilistic inference is done in [28] which produced high-performance algorithms for PRM, using d-separation. The authors of [29] analyze and describe the limitations of sectioning Bayesian networks. Another step for optimization of Bayesian inference and model construction was made in [30], introducing a general framework for canonical models. Its primary objective was the simplification of complex Bayesian models, especially those in which nodes have many parents.

2.1.4 Transforming probabilistic networks to increase run-time performance

Darwiche *et al.* [31,32] proposed compilation to Arithmetic Circuits (AC) to accelerate problem resolution time using BNs. ACs contain numbers, add and multiply operators; they are the standard model for computing polynomials. In probabilistic reasoning domain, ACs are an optimized structure to answer queries like "what is the most probable cause of a particular failure", "why a system manifests such a state", "why a failure occurred?". As Darwiche *et al.* propose, the probability distribution induced by a Bayesian network can be represented using a multilinear function (MLF). MLF is represented by Arithmetic Circuits, in which sum operators are changed to max operators. We call this representation Maximizer Circuit. An example circuit is presented in Figure 2.2.

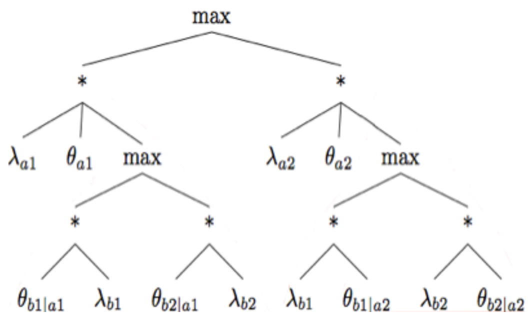


Figure 2.2: Example Maximizer Circuit. Comparing to AC, sum operators are changed for max operators. Lambdas are evidence indicators, and thetas are probability values.

The authors of [33] describe a successful deployment of Arithmetic Circuits in the diagnosis of spacecraft electrical systems. Another application of precompiled BN in a diagnostic system is presented in [34] but, this publication explores small systems, and it does not consider large-scale RCA. In particular, authors neither consider replicated components nor their subgraphs in Bayesian network representation. However, Arithmetic Circuits present numerical problems for huge systems that have millions of nodes, and furthermore, compilation requires large amounts of memory.

2.2 Graph representation of a system state and graph similarity for diagnostics and root cause analysis

In this Section we focus on work related to Graph representation of a system state, use of graphs for root cause analysis and root cause classification. Then, we move to the area of diagnosing a system state with partial or no knowledge.

2.2.1 Graph-based systems for root cause classification

Monitoring and logging systems are responsible for providing full observability of a system state, which is one of the most important inputs for a root cause classification system. Current research in this field is focused on dealing not only with the huge size and complexity of information encoded in logs but also with fault tolerance and use of partial information [35]. Usually, operators use sources of information in a troubleshooting process separately, *e.g.*, metrics and logs. Diagnostic tools do not combine well descriptive data such as text messages with metrics. One of the best ways to do so is by utilizing a graph representation of a system state. Constructing proper graph representation allows for anomaly detection and diagnostics [36]. Graph-based approaches are widely used for root cause classification, detection and prediction of abnormal events and failures [37–41].

When we represent system states as graphs, we can compare the encoded state by evaluating the similarity between them [42, 43]. An important contribution in the field of diagnostics via graph similarity is the work of Papadimitriou *et al.* [44] that evaluates graph similarities to find anomalies in the web. The authors consider different approaches for similarity evaluation which are limited to the topologies of compared graphs. Work on the similarity between different texts and logs is presented in [45, 46] and it is widely used for diagnostics of IT systems. Research of Putra *et al.* [47] includes graph-based text similarity evaluation. Other important work on utilizing similarity between logs that are used for diagnostics can be found in [48–50].

2.2.2 Performing diagnostics without exact knowledge of system failures

Diagnostic systems can gather the knowledge in one domain and reuse this knowledge for diagnosing similar systems with symptoms in similar knowledge domains. Generally, we call this type of use of knowledge *transfer learning* [51] and the *heterogeneous transfer learning* when the knowledge comes from different systems [52]. In this thesis, we focus on a scenario of the *transductive transfer learning*, where the data is labeled in the source domain, but not in the target knowledge domain. According to this area, one of the paths to deploy transfer learning in diagnostic systems is to apply similarity measures between a diagnosed state and the abnormal state to be diagnosed.

2.3 Model-driven prediction of failures in distributed systems

In this Section we explore work related to workload and failure analysis of supercomputers and HPC environments. We study research on prediction of failures and different actions taken for prevention. Then, we move to the use of logs for diagnostics and failure classification.

2.3.1 Predicting and preventing failures in High-performance Computing

Authors in [53] describe the role of software in failures occurring in data centers. Software problems in an OS, middleware, application, or the wrong configuration, *e.g.*, underestimated resources, cause the majority of job failures in HPC workload [54, 55]. The authors of [56] discover the correlation between failures, and different characteristics of supercomputer operations, such as node usage, last state of a job, and hardware metrics. This research explores state sequences from the perspective of a node. The authors perform job-oriented analysis only to point users with a high failure rate. Analysis of logs and the rate of failed jobs allows detecting slow-downs and targeted failures [57]. Authors in [58] propose a message-based prediction system for failures in a data center. Recently, the authors in [59] characterized workload in an HPC environment with the main goals to find patterns across different applications and disciplines. Latest work presented in [60] analyzes failures of the Oak Ridge supercomputer. The authors describe hardware reliability, correlate failure types, and investigate failure trends across time and space. However, leveraging user history for prediction of failed jobs and learning application workload patterns is not a primary focus area in these publications. Also, there are not many publications addressing the separate analysis of jobs and job steps. Survey on failures prediction confirms this [61].

There is much research on ML used for data center maintenance for either prediction or classification problems [62–65]. For instance, research in [66] uses dynamic association rules to predict failures in the Blue Gene. The authors of [67] focus on predicting failures in computing nodes, and as a reaction, redirecting a job to another set of nodes. Another possible action is checkpointing, and the authors of [68] investigate the optimal policy to reduce the trade-off between checkpoints frequency and mean time between failures. The authors in [69] use power and temperature metrics to predict errors in GPU clusters via neural network (NN) model. Recently, decision trees are implemented for failure prediction in HPC domain [70]. The proposed algorithm identifies the causes of failures, performing better comparing to other SoA techniques.

Despite the popularity and progress in ML algorithms and software, the area of prediction of the final HPC job states through accurate modeling of power series seems to be unexplored. Power series can be used to depict load of a node, since the electrical power of a node is proportional to the CPU load. It is helpful to use power series in many HPC environments, especially while CPU load is not acquired in order not to cause any inference with monitoring agents and a computing node performance. Power series depicting load of many nodes utilized by a particular job is a multivariate time series, while power series for one node only it is called univariate time series. The focus of most of the work is put on predicting failures per hardware unit, rather than learning workload patterns of failed jobs. The complexity of IT systems and their dynamic structure are one of the main obstacles to create accurate models. The authors in [71] propose power modeling techniques via Petri networks, to estimate power consumption. Also, the work presented in [72] reports research on power profiling in HPC environments. The authors discuss application network architecture, performance, and scalability in the dimension of power consumption, and they propose a system for accurate power monitoring.

2.3.2 Modelling system behavior through logs

The logs of an IT system are a valuable source of information used for data-driven diagnostics and prognostics of a system state. A usual method of working with logs, it is exploring the statistics and the occurrence of a set of key terms using log parsers, indexers, and miners. The authors of a survey on data-driven techniques in computing system management [73] claim that to realize the goal of self-management, systems need to automatically monitor, characterize, and understand their behaviors and dynamics; mine events to uncover useful patterns, and acquire valuable knowledge from historical log/event data. Fundamental knowledge of diverse approaches of error log processing is found in [74, 75]. Some simple mining methods include log key terms occurrence correlation [76], and modeling a multithreaded system behavior through graphs or sequences representing system calls. For instance, the authors of [74] deal with the problem of failure prediction through clustering similar log sequences. They propose an algorithm to assign the source of failures to logs, using Levenshtein's edit distance.

Recently, a considerable part of work on automated diagnostics is performed with the help of Artificial Intelligence and Machine Learning. The DeepLog system [77] is one of the most significant contributions in this field. The authors propose a system for anomaly detection and diagnosis which is based on deep learning. The performance and accuracy of the solution are high. However, to use it, there is a necessity of defining metadata. For this reason, the solution has limited usability regarding full automation. Authors of [78] propose an approach to mine time-weighted graphs from logs with many threads running. The solution evaluated in cloud environments performs with high recall and precision. Authors in [79] use casual inference to diagnose network failures. They mine casual graphs from logs, considering connected devices in a graph. One of the conventional approaches to deal with log preprocessing and comparison is transforming log entries to vectors, using the Word2Vec algorithm [80, 81]. A recent attempt to leverage Word2Vec for root cause classification is described in [82]. The authors propose a method for processing logs with a Word2Vec model and then using a Bayesian classifier.

CHAPTER 3

LARGE AND DISTRIBUTED IT ENVIRONMENTS DEMANDING HUGE OPERATOR EFFORT IN MAINTENANCE

In this chapter, we describe IT environments which need significant human effort for diagnostics and troubleshooting. We use these environments as use cases for different research problems. We make a choice based on the following criteria. Firstly, we want to cover a wide spectrum of use cases, that is why we do not limit research to one environment. Secondly, we want the chosen environments to imitate different problems of diagnostic systems and operators: scalability, dynamism, partial knowledge and predictability. Also, we choose the ones which are not only critical today but are essential for the future development of the IT sector. The chosen platforms are: 1) a supercomputer, that is widely described in Section 3.1; 2) an Internet of Things stack described in in Section 3.2; and 3) a Big Data cluster and a system of microservices based on containers, that is described in Section 3.3. At the end of each section, we emphasize typical problems which require a lot of maintenance in a particular environment.

3.1 An HPC environment: The Mistral supercomputer

The following Section presents a short description of a supercomputing environment used for research presented in Chapter 6. The German Climate Computing Center manages the supercomputer Mistral[1] that is ranked as 55th most powerful on the world as of June 2018[2]. The HPC system has a peak performance of 3.14 PetaFLOPS and consists of approximately 100,000 computing cores, 266 Terabytes of memory, and 54 PiB of Lustre file system. Also, DKRZ maintains an automated tape data archive with the capacity of around 500 PiB. The supercomputer workload is generated by a variety of applications and simulators used in areas such as climate science, geology, and natural environment. Example applications include simulation of Hamburg ecosystem and its pollution, North Sea ecosystem, and Earth climate change.

In details, the supercomputing facility contains 3336 computing nodes placed in 47 racks, and about 90 special nodes dedicated for maintenance activities, data pre-processing, post-processing and advanced visualizations. Understanding the data center structure and topology is essential for the process of creation of diagnostic models. This knowledge helps to determine the structures from which we need to collect data and understand discovered phenomena and trends. Computing nodes are placed in racks. Majority of the racks are homogeneous having mounted the same 72 blades. Each rack encloses maximum 4 chassis, with the maximum capacity of 18 blades per chassis. Each of these units: rack, chassis, and a node are managed by a dedicated controller. There are rack management controllers, chassis management controllers and blade management controllers. In this thesis we utilize data coming from blade management controller.

The computing blades installed in the Mistral are Atos Bull B720 containing two computing nodes. These nodes can contain either Intel Xeon 12C 2.5 GHz (Haswell) or Intel Xeon 18C 2.1 GHz (Broadwell). Table 3.1 presents the number of blades by type and the number of racks containing only one type of a blade. For instance, B720_24_64 stands for Atos Bull B720 with 24 cores and 64 GB of RAM. For example, in the Mistral system, inventory tables contain detailed information about all the installed equipment, its interconnections, management controllers, and localization. For instance, in Table 3.2 we present a summary of equipment located in Mistral facility.

In the data center, resource allocation and accounting are maintained using Slurm[3]. This open

[1]https://www.dkrz.de/up/systems/mistral
[2]Ranking TOP500 June 2018 https://www.top500.org/system/178567
[3]https://slurm.schedmd.com/

source Resource and Job Management System manages reservations performed by users, schedules jobs and accounts the consumed resources. First, it allocates exclusive and non-exclusive access to resources (compute nodes) to users. Second, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work. Optional plugins can be used for, *e.g.*, accounting, advanced reservation, topology optimized resource selection, resource limits by a user or bank account. Mistral computing nodes are divided into several Slurm partitions: development, pre- and post-processing, test, production. In this thesis, we analyze data from the production partitions.

According to the Mistral supercomputer, the most frequent errors are these associated with internal job problems: source code bugs or wrong input format. The external causes are usually independent for a job and in the majority, they are associated with Lustre system. Maintenance team spends most of the troubleshooting time on Lustre problems, they might also debug particular jobs in a small-scale system. Hardware problems, *i.e.*, network, cooling infrastructure and computing nodes are sporadic.

Table 3.1: Computing nodes in Mistral

Node type	**Total quantity**	**Homogeneous racks, 72 blades**
B720_36_64	1454	20
B720_24_64	1404	19
B720_36_128	270	3
B720_24_128	110	1
B720_36_256	50	-
B720_24_256	48	-

Table 3.2: Equipment in the data center other than nodes

Equipment	**Quantity**	**Description**
B720-bmc	3336	Blade Management Controller
BC-DLC-cmc	186	Chassis Management Controller
BCM-56224	186	Broadcom BCM-56224 Eth. Switch
FSB	186	Infiniband Switch
HYC-DLC-hyc	94	Hydraulic Chassis
SSU-CE9000	62	Disk Array
CPC-DLC-cpc	47	Cooperative Power Chassis
R-DLC-rmc	47	Rack Management Controller
SX6025	45	Infiniband Switch
ICX-6610-24	42	Eth. Switch
NSR423e4i-bmc	20	Blade Management Controller
NSR421e4-bmc	17	Blade Management Controller
MPX2-5530	12	Raritan, Power Distribution Unit
NODE-Xyratex-bmc	8	Blade Management Controller
NSR424e4-bmc	8	Blade Management Controller
3750-48	8	Cisco 3750-48 Eth. Switch
HYDRA-colddoor	6	Cold Doors
NSR425e4-bmc	4	Blade Management Controller
ADU	4	Disk Array
SX6536	3	Infiniband Switch
2960S-48	3	Cisco 2960-S-48 Eth. Switch
3750-24	2	Cisco 3750-24 Eth. Switch
SX6036	2	Infiniband Switch
CE2700	1	Disk Array
SG300-52	1	Cisco SG300-52 Eth. Switch
SG300-28	1	Cisco SG300-28 Eth. Switch
2960S-48-1	1	Cisco 2960-S-48 Eth. Switch

3.2 Internet of Things stack

The rapid development of the Internet of Things (IoT) and increasingly widespread use of mobile and smart devices generating frequent data collection and exchange needs are forcing organizations to change the way they engage customers, develop and deliver new products and services. Consequently, data analytics is ubiquitous, bringing intelligence to every process [83]. According to Cisco, IoT will unleash \$19 trillion in new profits and cost savings globally in the next decade [84]. Besides, global data center traffic will grow nearly 3-fold from 2014 to 2019, and by 2019, more than 86 percent of workloads will be processed by cloud web services in data centers [85]. Regarding work about Big Data and IoT frameworks in [86], data coming from IoT systems, *e.g.*, smart cities, are characterized by a high diversity of their structure, a high degree of variability, high velocity, and huge volume. Furthermore, data are transformed and analyzed at different layers of a system, spreading from preprocessing in the sensor microprocessors to data centers running data mining and deep learning applications.

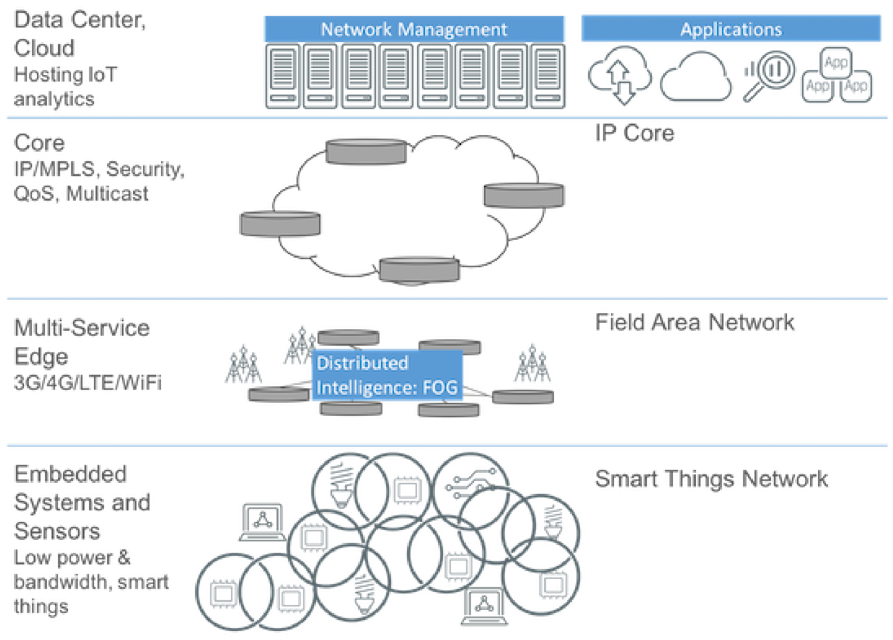


Figure 3.1: Overview of IoT computing model in reference to [1] (Permitted for use)

A growing amount of data and the demand for their processing bring about new approaches and paradigms in network and data centers infrastructure. Measurements and data coming from IoT devices are not only processed in the cloud since the infrastructure and processing capabilities can be insufficient. Fog computing handles the needs of, *e.g.*, geographical distribution of resources, real-time communication, incorporation with large networks. Through this paradigm, part of the processing is done by edge devices or clouds closer to data sources, resulting in less latency and bandwidth usage [87]. An example IoT stack can be seen in Figure 3.1.

For an efficient monitoring, troubleshooting and management of enormous IoT environments it is necessary to provide a robust root cause analysis (RCA) mechanism which is scalable and tractable enough to perform fast diagnosis on the whole system and will find explanations of the problems whether they are located in the neighborhood of the particular device, other processing tier or they are compound.

An essential aspect of the maintenance of IoT infrastructures is the connectivity between the devices. The connections can be interrupted not only because of devices changing the localization and consequently being out of network coverage. Devices can also disconnect due to power problems.

3.3 Cloud and on-premise Big Data environments

Whether cloud or on-premise, Big Data applications deliver extensive business analytics and functionalities with streams of data, large data volumes and different types of data. Big Data is used to discover new correlations and patterns, which lead to valuable business insights.

In this thesis, we focus on two types of infrastructures that are: on-premise and cloud one. On-premise infrastructure is linked to a scenario where an entity holds servers on site, maintaining them physically and virtually. The cloud-based infrastructure represents emerging technologies of microservices and scalable infrastructure where the resources are adjusted to the user demand.

3.3.1 On-premise Big Data cluster

We run Apache Spark and Hadoop on the following infrastructure. The cluster system comprises:

- 5x server: 32 GB RAM, AMD Opteron(tm) Processor 6168 (12 cores, 1.9 GHz), equipped with IPMI card and running Ubuntu OS
- Switch D-link DGS-1210-48
- 2x Power Analyzer ZES Zimmer LMG450. The device is a 4-channel power analyzer mounted in a rack and connected between each power supply and servers and the switch.

3.3.2 Cloud based infrastructure and microservices

Nowadays, microservices and containers are used as a cost-optimized, dynamic infrastructure for running web applications, databases, stream processing applications and others. We use Grid5000[4] to run Docker[5] containers. We use different software architectures and solutions. We describe them later on in Section 5.4 where they are introduced for the experiments.

One of the most frequent and essential problems experienced in the distributed infrastructures is a *dead node* problem. Usually, there might be many independent causes associated with a failure, both hardware and software related.

[4]https://www.grid5000.fr/
[5]https://www.docker.com/

CHAPTER 4

SCALABILITY AND DYNAMISM IN ROOT CAUSE ANALYSIS AND DIAGNOSTICS BY MEANS OF PROBABILISTIC REASONING

The number of applications of the Internet of Things, Cyber-Physical Systems, fog computing and large data centers hosting cloud and web services is rapidly growing. The primary challenge of maintenance and operations is not only the scale of these systems, but also their distribution, and dynamism.

In this Chapter, we present 2 new RCA systems. First, we propose a new root cause analysis system. This system leverages the fact that the IT systems mentioned above usually contain a lot of repeated elements. The proposed system is based on Bayesian reasoning and provides a novel cache-based mechanism. We use Arithmetic Circuits as a computing structure which facilitates the reasoning process. Thanks to the fact that ACs can be split into subparts, it enables the reuse of previous computations to speed up the inference. The presented solution provides a fast RCA that takes milliseconds when the system model changes, and it does it without the necessity to re-transform the whole BN again into the ACs.

Then, we introduce ABRCA, an actor based root cause analysis which is a distributed, decentralized and fault tolerant reasoning algorithm. It can use resources of the devices making up the system to perform calculations that are necessary for root cause analysis. We evaluate the proposed algorithms on the diagnostic models, which consists of millions of nodes, and simulate IoT devices exchanging data with a data center. For ABRCA we simulate the behavior of the devices comprising an IoT environment. Results show that the system can perform an order of magnitude faster, using fewer resources.

Our contributions are:

- Root cause analysis system based on split Bayesian networks and their transformation that are Arithmetic Circuits. The system leverages the fact that in large IT systems there are many repeated components, it manipulates Arithmetic Circuits and caches the partial results and structures. The system scales to millions of nodes in a diagnostic model. (Section 4.2)

- Actor based root cause analysis system. The system uses the distributed resources of the devices making up a system for root cause analysis calculations. The actor based model allows for the implementation of different policies in case of failures. (Section 4.3)

4.1 New root cause analysis methods for large-scale systems

Today, the growing popularity of the Smart City vision and its integration [88–90] with technical domains such as the Internet of Things (IoT) and Cyber-Physical Systems (CPS) generates new challenges. The important ones are management, troubleshooting, and the control of these IT environments [91]. The software industry is forced to look for new solutions and algorithms for diagnostics because of the huge complexity and the size of networks comprising intelligent devices. In particular, the diagnostic solutions for these environments should be extraordinarily fast, accurate and automatic, needing little human effort for operation.

Usual characteristics of Cyber-Physical Systems and IoT is that they consist of various intelligent devices continuously connecting and disconnecting, *e.g.*, because of switching a used network. This type of behavior may be caused by device changing its localization, some failure or the loss of the connectivity due to external conditions. As a consequence, diagnostic models are continuously changing, as they should reflect the current structure of a system. Also, these models can change when the new devices connect to the system, the hardware upgrades or the structure of a physical network changes.

While the considered systems change their structure rapidly, they push existing solutions to their limits and make them difficult to react in time. For instance, a centralized diagnostic system might not be reliable enough to provide fast and accurate troubleshooting. Also, the geographical dispersion of these systems, in particular, CPSs, and their limited bandwidth reduces the system capability of performing real-time diagnostics, especially when a system loses the connectivity with a part of it. Thus, the analysis cannot frequently be performed with a satisfying level of accuracy, and the use of little resources, *i.e.*, having a memory utilization at the level of hundreds of megabytes for BN with more than a million nodes.

Scalability and dynamism of diagnosed systems become the primary challenge to perform root cause analysis effectively. We create a framework which can provide robust and automatic RCA in distributed environments. We propose a new root cause analysis system that leverages the fact that systems mentioned above usually contain a lot of repeated elements. The system is based on Bayesian reasoning and provides a novel cache-based mechanism. We use Arithmetic Circuits as a computing structure. Thanks to the fact that ACs can be split into subparts, it enables the reuse of previous computations to speed up the inference. The presented solution provides a fast RCA when the system model changes, without the necessity to recompile the whole BN. We evaluate our algorithm on a diagnostic model containing millions of nodes. The model simulates Internet of Things devices exchanging data with a data center. In the evaluation we show that the proposed system performs an order of magnitude faster.

The proposed system performs reasoning using Arithmetic Circuits which are compiled from Bayesian networks and are much faster in the runtime. Thanks to the use of AC, we make it possible to manipulate AC computations and structures while a diagnostic model changes, without recompiling BNs. We are also able to reuse compiled structures for different instances of the same diagnostic model. Our contribution results in less memory footprint, faster diagnosis and better scalability of the diagnostic process compared to the use of other conventional systems, *e.g.*, based on Case-Based Reasoning (CBR) [25]. In Section 4.2 we describe details of the proposed fast RCA for large environments.

Additionally, we introduce ABRCA, an actor based root cause analysis which is a distributed, decentralized and fault tolerant reasoning algorithm. We focus on the aspect of **dynamism**: the system size and connections between devices are frequently changed. The proposed solution uses resources of the devices making up the system to perform calculations that are necessary for root cause analysis. We describe this contribution in Section 4.3.

Then, we evaluate the proposed approaches. In Section 4.4, we evaluate RCA based on AC for a large simulated environment of Internet of Things devices exchanging data with a data center. Then in Section 4.5 we evaluate the performance of ABRCA in the conditions of devices connecting and disconnecting simultaneously.

4.2 Root cause analysis for large networks of devices through splitting, transforming and reusing Bayesian networks

In this Section, we explain fast RCA method. As we know from the previous Section, root cause analysis is a widely used method to identify causes of failures in a system. Tracking a causality between events allows for the determination of the causes of failures. One of the successful frameworks used for RCA is a probabilistic network method like Bayesian networks. One of the most significant advantages of using Bayesian networks is that they perform accurate diagnosis even if information about the system state is not complete. However, Bayesian Networks are expensive to calculate and update, even when using improvements such as pre-compilation through Arithmetic Circuits. Such scenario is usual in IoT and CPS environments where structure and network coverage changes. Also, the aspect of many system structure changes will be more common, as long as the use of emerging technologies, such as Software Defined Infrastructure [92, 93] keeps growing. To perform accurate root cause analysis, reasoning should be performed considering a large number of statistics, dependencies, and observations. Including all these data results in the large size of a diagnostic model that is a network of millions of nodes, and greater computational complexity.

Furthermore, the diagnostic system should be flexible enough to cope with these changes, reducing recalculations as much as possible. Considering research in [94] on RCA using large Bayesian networks, two important conclusions can be drawn: (i) network can be divided into clusters, *i.e.*, sub-networks which reduce calculation complexity and (ii) the root cause is usually in the neighborhood of the observed failures.

In particular, we propose to take the steps of Algorithm 1 to calculate the root cause of a particular system state.

Algorithm 1 Steps of the proposed root cause analysis method

```
1: Use Bayesian networks to create diagnostic models of a system
2: Create a model of a diagnosed system with the Probabilistic Relational Model paradigm
3: Identify repeatable system elements (objects) and split them for separate models (clusters)
4: Transform unique models defined as Bayesian networks to models based on Arithmetic Circuits
5: Evaluate these models given observations for each object. During the evaluation leverage cache
   mechanism.
6: Join results of evaluations according to the hierarchy and defined diagnostic model of the entire
   system. Leverage the construction of the Multi-linear functions (MLF).
```

In Figure 4.1 we present a high-level scheme of a traditional approach of root cause analysis. The system processes the stream of events (1) and uses defined model of the whole system to perform root cause analysis. In Figure 4.2 we present the system based on traditional approach and compiled models. The offline compilation is an additional step comparing to the system from Figure 4.1. A compilation module (3) provides transformed diagnostic models to be used in an online evaluation. It allows handling expensive resource calculations out of the runtime. Then, in Figure 4.3 we present setup of the proposed RCA system that realizes steps described in Algorithm 1. It splits models of system objects, compiles them separately and composes results to formulate the final hypothesis. Note that in Figure 4.2 system description is to transform the diagnostic model of a whole system. In contrast, in Figure 4.3 this description is used at the last step of calculating the RCA, to aggregate results from split models.

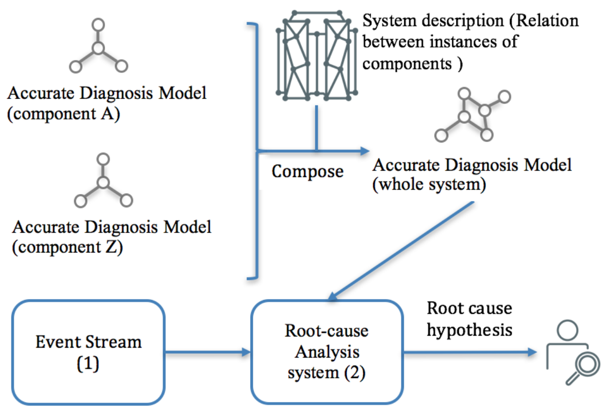

Accurate Diagnosis Model (component A)
Accurate Diagnosis Model (component Z)
System description (Relation between instances of components)
Compose
Accurate Diagnosis Model (whole system)
Event Stream (1)
Root-cause Analysis system (2)
Root cause hypothesis

Figure 4.1: Root-cause analysis traditional system diagram

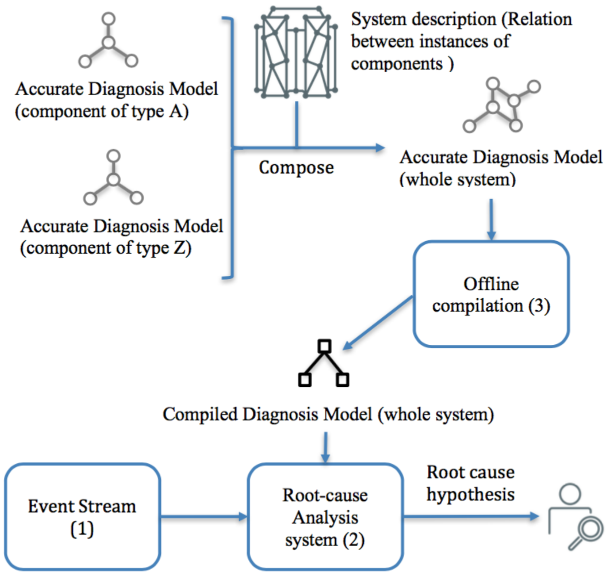

Accurate Diagnosis Model (component of type A)
Accurate Diagnosis Model (component of type Z)
System description (Relation between instances of components)
Compose
Accurate Diagnosis Model (whole system)
Offline compilation (3)
Compiled Diagnosis Model (whole system)
Event Stream (1)
Root-cause Analysis system (2)
Root cause hypothesis

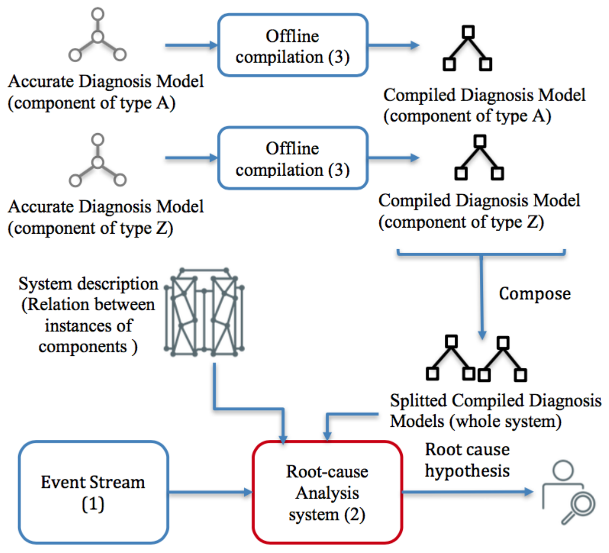
Figure 4.2: Traditional diagnosing system based on a compiled model

4.2.1 Bayesian networks and Arithmetic Circuits

According to work of Judea Pearl [6]:

Definition 1 *Let* $\boldsymbol{U} = \{\alpha, \beta, ...\}$ *be a finite set of elements,* e.g., *variables, and* $\boldsymbol{X},\ \boldsymbol{Y},\ \boldsymbol{Z}$ *stand for three disjoint subsets of elements in* $\boldsymbol{U}$*. Then, a dependency model* $\boldsymbol{M}$ *it is a rule that assigns truth valuers to the three-place predicate* $I(X,Y,Z)_M$.

Definition 2 *A Directed Acyclic Graph (DAG)* $\boldsymbol{D}$ *is said to be an* ***I-map*** *of a dependency model* $\boldsymbol{M}$ *if*



Accurate Diagnosis Model (component of type A)
Offline compilation (3)
Compiled Diagnosis Model (component of type A)
Accurate Diagnosis Model (component of type Z)
Offline compilation (3)
Compiled Diagnosis Model (component of type Z)
Compose
System description (Relation between instances of components)
Splitted Compiled Diagnosis Models (whole system)
Event Stream (1)
Root-cause Analysis system (2)
Root cause hypothesis

Figure 4.3: System diagram of the proposed system solution

*every d-separation condition displayed in **D** corresponds to a valid conditional independence relationship in **M**.* i.e., *if for every three disjoint sets of vertices X, Y, Z we have:*

$$< X|Y|Z >_D \implies I(X, Y, Z)_M$$

Definition 3 *Given a probability distribution **P** on a set of variables **U**, a Directed Acyclic Graph*

$$D = (U, \overrightarrow{E})$$

*, where $\overrightarrow{E}$) is a set of directed edges, is called a **Bayesian network** of **P** if **D** is a minimal **I-map** of **P**.*

Definition 4 *A multi-linear function (**MLF**) for a **Bayesian network** with variables A and B is represented as follows*

$$f = \sum_{b} \prod_{ba\sim b} \lambda_b \theta_{b|a}$$

where λ_b denotes evidence indicators for B and $\theta_{b|a}$ stands for parameters associated with its conditional probability depending on the value of A.

Definition 5 *An arithmetic circuit over variables **U** is a rooted, DAG whose leaf nodes are labelled with numeric constants or variables in **U** and whose other nodes are labelled with multiplication and* *addition operations. The size of an arithmetic circuit is measured by the number of edges that it contains.*

As explored earlier in Section 2.1, inference in Bayesian networks is resource expensive. Structures which can be used alternatively are Arithmetic Circuits. Arithmetic circuits are introduced in [31], and they are based on Multi-linear functions (MLF) which is created from a given Bayesian network. In fact, an AC describes probability function of a BN, in a manner which facilitates calculations during Bayesian inference.

The transformation of Bayesian networks to AC is done through the process of compilation. These transformations do not cause loss of diagnostic accuracy, either sensitivity of the original model and can be evaluated much faster. Then, compiled AC are evaluated to provide results of calculations. The only disadvantage of using Arithmetic Circuits directly is that their size and topology are not making it easy to be interpreted by an operator.

4.2.2 Most Probable Explanation as the result of root cause analysis

Discovering the root cause in the model based on BNs, it means to solve the problem of a calculation of a Most Probable Explanation (MPE), which is defined as follows.

Definition 6 *Most Probable Explanation (**MPE**). Given is evidence e which represents set of all variables that are determined (observations). Computing an **MPE** is a problem of finding such an explanation $W = w^*$, where W stands for the set of all variables considered, including those in e - given evidence, in the **Bayesian network**, that maximizes the conditional probability $P(w|e)$ [6]*

$$P(w^*|e) = \max_w P(w|e)$$

The calculation of MPE is intractable and remains NP-hard, even if all variables are binary and both outdegree and indegree of the nodes are at most two [95]. The problem can be partially solved by limiting the size and the complexity of the network used for calculations. Our proposition is the precompilation of the subnetworks of the replicated elements. Then we join them in a specific manner that allows reusing computations and as a consequence an acceleration of the diagnosis of a huge system.

In the proposed approach, we leverage the fact that, an Arithmetic Circuit can be easily transformed to the *maximizer circuit*, which is a structure designed to calculate **MPE**. This structure compared to AC performs *max* operations instead of *add* ones. The complexity of the AC compilation process time, as well as the inference, is $O(n\,exp(w))$, where n stands for the number of variables and w for the tree-width of the transformed Bayesian network.

4.2.3 Transforming and splitting models

The proposed method follows the concept presented and proved in [25]. The root cause analysis is realized through approximate reasoning performed on subnetworks of a diagnostic model. An example Bayesian network is presented in Figure 4.4. The network has three nodes and each of the nodes has two possible states $B_1 : \{b_{11}, b_{12}\}, B_2 : \{b_{21}, b_{22}\}$ and $A : \{a_1, a_2\}$. These states could have any arbitrary meaning like b_{11} being "B_1 is working fine" and b_{12} being "B_1 has a problem" for instance. An AC (maximizer circuit) created from this network and designed to calculate **MPE** can be seen in Figure 4.5.

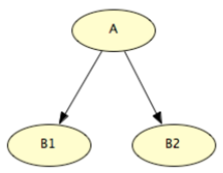


Figure 4.4: The example Bayesian network to transform into AC

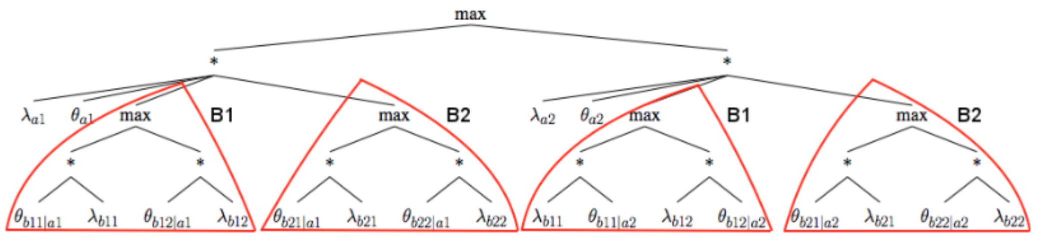


Figure 4.5: The Arithmetic Circuit for the example Bayesian Network with marked parts corresponding to the B_1 and B_2 nodes

If states and conditional probabilities of nodes B_1 and B_2 are the same, there is no necessity to compile the whole BN from Figure 4.4, but only consider the one shown in Figure 4.6 and then aggregate computations from replicated nodes during the evaluation of the AC. The result of transformation BN from Figure 4.6 into the AC can be seen in Figure 4.7. In Figure 4.7 the parts of the AC that are replicated if more nodes of type B are added to the Bayesian Network from Figure 4.6 are marked.

It can be seen that, in the worst-case scenario, which is a system without replicated elements, the complexity of the AC size grows as mentioned before. The proposed RCA framework prepares an Arithmetic Circuit for each object type in a diagnosed system. Thus, the network representing a model has lower complexity. Instead of compiling the whole system network, the compilation is only invoked once for each component class of the system. Specifically, it means that if a system is composed of

1000 components of the same type, only a single compilation for the component would be required, leveraging the fact that most complex systems have a large number of replicated components.

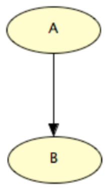


Figure 4.6: The Bayesian network example

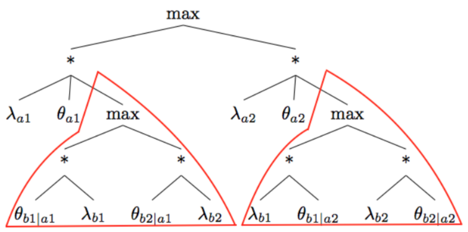


Figure 4.7: The AC with multiply and max nodes for calculating MPE in Bayesian network on Figure 4.6 with marked parts corresponding to the B node

We assume that the preliminary model of a system is not split. In such scenario, to include connections between components, specific Bayesian network nodes which are responsible for interconnection are cloned from a parent component and placed in a referencing object (child component). This particular step is illustrated in an example model having two components, see Figure 4.8. After this processing step, *Component 2* contains one duplicated node coming from *Component 1*, as shown in Figure 4.9.

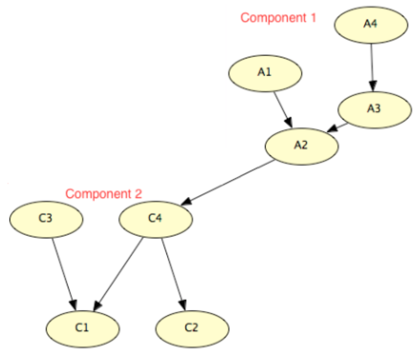


Figure 4.8: Diagnostic model in form of BN, for two components of different types

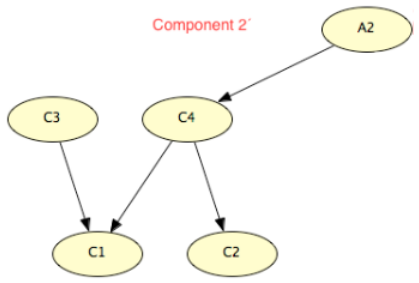


Figure 4.9: BN of *Component 2* after its transformation, prepared for compilation into AC

Below, we present Algorithm 2 that performs RCA evaluation of models and aggregation of the results. The input consists of (i) compiled diagnostics models - AC for each component type, including a reference to the external nodes, (ii) system instances schema defining dependencies between specific nodes of instances, number of instances of each component and their connections and (iii) set of evidence (observations of a state). As a result, we receive marginal probabilities for each variable in the diagnosed system.

We use the following notation:

- I : single instance of a component
- $I.S$: nodes in instance I which are referenced to by other external nodes from other instances (S stand for slots)
- $I.P$: nodes in instance I which are external nodes from other instances, thus in instance I' they are cloned (P stand for plugs)
- $I.N$: internal nodes (including $I.S$)
- $s.A$: aggregated value of a node s, which is referenced by external nodes
- $p.v$: value of a node p

Algorithm 2 Pseudo code presenting an algorithm of calculation of a MPE for a given diagnostic model. MPE is considered as the final outcome of root cause analysis

```
Input: Compiled models $M$, instances schema: $\Pi$, set of evidence $e$
Output: set of MPE for the whole diagnosed system
    Initialisation : start with instances where I.S = $\emptyset$
 1: for all devices $I$ in $\Pi$ do
 2:    key := (type($I$), e, weights)
 3:    for all $s$ in $I.S$ do
 4:       assert $s.A$ aggregated all summands
 5:       if not global cache contains value for key then
 6:          for all node $s$ in $I.S$ do
 7:             weight := $s.A$+log($s.probability$)
 8:             add s.id $\rightarrow$ weight to weights map
 9:          end for
10:       else
11:          result := cache[key]
12:       end if
13:    end for
14:    result := evaluate M[I] with e and weights
15:    put key $\rightarrow$ result in cache
16:    put $I.N$ $\rightarrow$ result in the MPE set
17:    for all nodes p in I.P do
18:       let I' stand for an instance where is a node that p was cloned from
19:       assign values p.v to its referenced node I'.S for aggregation
20:       nodes I'.S aggregate received value incrementally with an accumulated I'.S.A
21:    end for
22: end for
```

4.3 Distributed root cause analysis through probabilistic self-diagnostics for dynamic systems

In this Section, we propose an innovative diagnostic system to monitor a distributed and dynamically changing environment. Actor Based root cause analysis (ABRCA) is an algorithm that performs RCA by using distributable probabilistic reasoning through an actor model [96]. By taking advantage of a supervision mechanism, the system can automatically handle faults that occurred while performing root cause analysis. An exact policy on how the diagnostic process should be changed when the faults have occurred can be defined per each device type.

The first significant novelty of the proposed system is that it leverages already available computational resources of the devices which make up the diagnosed system. It gives the system the ability to self-diagnose. Consequently, this leads to the system decentralization and enables to diagnose it in many critical situations, when information from the whole system cannot be retrieved. For instance, while a connection problem or power outage is occurring, it can be possible to diagnose the system at a level of subnetworks successfully. Also, despite a global failure in the system, local causes can be found based on the observations and measurements coming from the neighborhood of devices. The system is proven to work in a distributed and large-scale simulated environment, where devices exchange messages asynchronously and in parallel.

4.3.1 Actor based root cause analysis (ABRCA)

In this research, the diagnostic models are defined as split Bayesian networks, and the reasoning mechanism is implemented using Arithmetic Circuits. So, each split BN contains nodes representing (1) internal states of a device, metrics, and failures; as well as (2) causality with a parent device. The results of AC evaluations are exchanged between the devices to keep calculations coherent. Also, it is possible to use other structures and evaluation algorithms, *e.g.*, message passing algorithm such as loopy belief propagation [97].

The first stage of ABRCA is a compilation of diagnostic models, defined as BNs, to ACs. This operation requires significant resources for computations. The second stage is an online inference in the compiled models, which outputs the MPE of the system state. A scheme of the proposed system working in an IoT environment can be seen in Figure 4.10. This distributed system includes modules responsible for the definition of a model, supervision, and evaluation. The model definition module stores current system structure, including the number of devices, their types, and interconnections. Another module, the evaluator coordinates RCA and enables communication with the devices. The supervision module works in each device's actor and controls the execution of calculations and reacts to failures which occurred during the runtime, *e.g.*, restarts calculations in the region of devices where the failure occurred. The behavior of the actor in case of failure is defined by supervising strategies, which specify what kind of action should be taken in case of calculation failure.

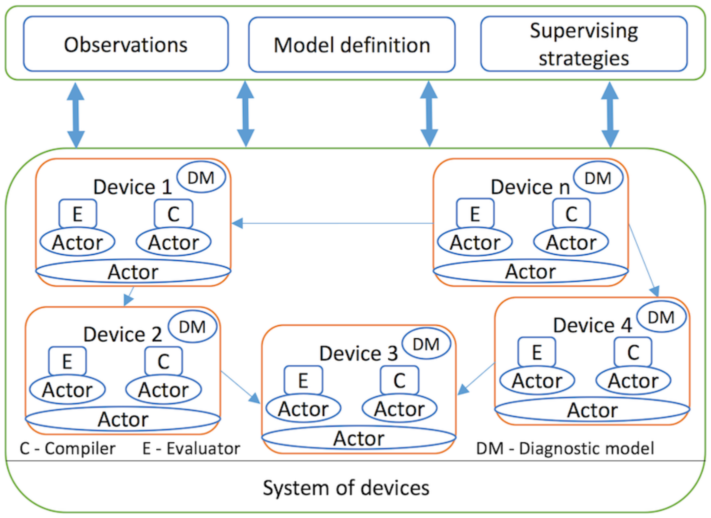


Figure 4.10: Actor based root cause analysis (ABRCA) system scheme

Decentralized modules of the system are actors responsible for calculations performed in the devices. The asynchronous communication between actors is based on instruction and calculation messages which are sent between them. Each device is associated with actor instances accomplishing a particular functionality, *i.e.*, model compilation, model evaluation, management and delegation of diagnostic tasks for the devices.

Using the actor model and its supervision mechanism results in the fault-tolerant solution during the computations of the calculations. Resources in each device are used to execute compilation and evaluation of Arithmetic Circuits, while regular processes are executed in the background. This approach enables to retrieve partial diagnosis, even if some components are not available. When devices connect and disconnect simultaneously, the system changes its diagnostic model through creation or destruction of the new actors. Once a new device connects, and its diagnostic model is already compiled, *e.g.*, the same type devices exist in the system, it reuses the cached model. Otherwise, it starts the compilation process. This event is communicated to the environment. Thus, other devices of a particular type can save their resources and wait for the compilation result which is eventually used for diagnostics. Besides, after a specified timeout, each device starts the compilation independently. This type of device behavior makes the system tolerant of either communication or calculation failures which might occur during the compilation process.

Described ABRCA increases the overall execution time of the RCA system in the case when the compilation time of a diagnostic model is much lower than a message delivery time. While only one device of a particular type compiles the Bayesian network, the communication bandwidth is highly consumed during the sharing of a result with other devices. On the other hand, when each device compiles the model independently, the resources are highly consumed, in contrast to the connection bandwidth which is less used. The ratio between devices compiling the model and waiting for the compiled one should be set according to system characteristics. These characteristics are associated with a cost of resources utilization, compilation time, which is related to the model complexity and the communication bandwidth.

Once the RCA is executed, something that happens either automatically or through an external request, devices perform inference in their models. All processes run asynchronously, so each device starts the inference at the moment of receiving the request and has all necessary inputs to perform calculations.

The communication between devices is designed as non-blocking. Once the message is sent, the device is not waiting for the answer and can process other requests. Messages are produced as soon as possible and without any dependency on the receiver state. For this reason, there is a necessity to provide a robust solution for their processing considering a correct time order and the receiver's internal state. Therefore, a stash component is used, so all messages that arrive at the device and are not processed at the moment are temporarily stashed in the queue. Thanks to this robust mechanism, none of the messages is lost or processed at an inappropriate stage of the algorithm.

We present Algorithm 3 which formulates activities performed by each actor in the system.

In Figure 4.11 we present the state machine diagram representing the behavior of a device actor. It depicts the evaluation of the diagnostic model of the whole system, whose result is the most probable root cause of the given system observation. The starting state is the one in which the model is not compiled yet, and the device has the connections established. The next states are associated with model compilation and evaluation. If the device has *children* devices, before the evaluation, it has to collect all the weights coming from these devices' calculations.

Algorithm 3 Pseudo code of different actions performed by ABRCA system while devices are connected, and a system performs RCA

```
Input: set of observations $O$, system scheme $S$, set $DM$ of Bayesian network diagnostic models for
    each device type
Output: MPE for the whole diagnosed system
    Initialisation :
 1: for all device $D$ in $S$ do
 2:     send Start message to $D$
 3: end for
    Message handling :
 4: switch (device $D$ in $S$ received message $m$):
 5:     case ($D$ received Start):
 6:         $D$ starts compilation of its diagnostic model $DM[D.type]$
 7:     end case
 8:     case ($D$ finished compilation):
 9:         $D$ sends compiled model $DM$ to shared (centralized) resources
10:         $D$ changes internal state for ReadyForEvaluation
11:         $D$ broadcasts CompilationFinished message to $\forall D' \in S : D'.t = D.t$
12:     end case
13:     case ($D$ received CompilationFinished):
14:         $D$ changes internal state for ReadyForEvaluation
15:     end case
16:     case ($D$ received RunRCA):
17:         $D$ sends Evaluate message to its engine actor $D.E$
18:     end case
19:     case ($D.E$ received Evaluate):
20:         if ($D.E$ collected all calculation weights $w$ from connected devices) then
21:             evaluate $D.E$ with $input = \{w, O[D]\}$
22:             $D.E$ sends messages with calculation results $w'$ to connected devices $D_C$ specified in $S$
23:         end if
24:     end case
25:     default:
26:         $D$ stashes $m$
27:     end default
28: end switch
```

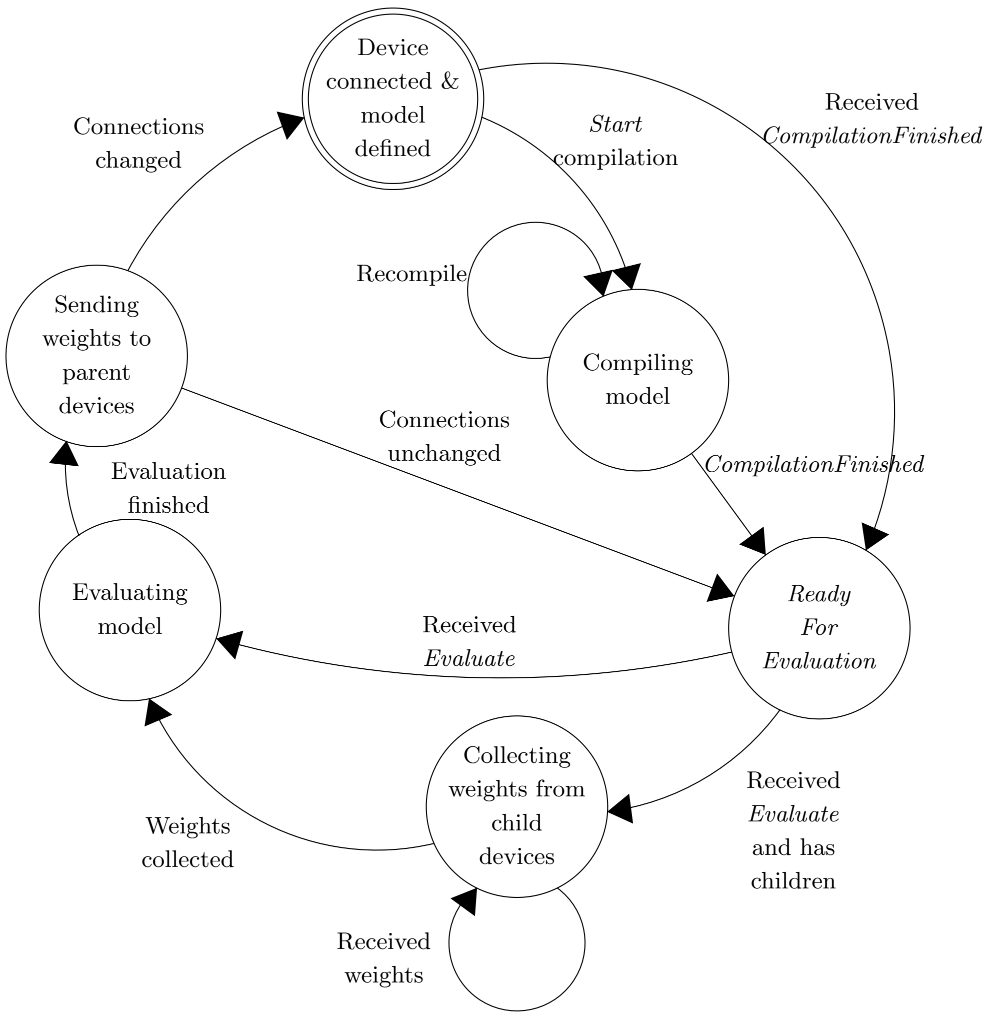


Figure 4.11: The state machine diagram representing behaviour of a device's actor

4.4 Evaluation: Exploring scalability of fast RCA

The first part of experiments aims to compare time and memory performance of the proposed fast RCA with the existing reasoning approaches and validate RCA accuracy. We use a diagnostic network that simulates large IoT network and data center environment, which is introduced in Section 3.2. It simulates well the scale and the complexity of a real environment. For references, we take conventional approaches that can be implemented in a simulator. Specifically, we compare our solution to (1) the compilation of whole BN of diagnostic model into one Arithmetic Circuit and (2) Case-Based Reasoning.

4.4.1 Implementation of the proposed system and methodology

We implement the fast RCA with Scala and Java, outputting a program to run in JVM. We use Ace 3.0 library for an efficient AC compilation and evaluation. We implement root cause analysis based on Case-based Reasoning using FreeCBR library. Since calculated probabilities in huge networks are small orders of magnitude, *i.e.*, 10^{-100}, it is necessary to use logarithmic calculation space, to avoid interrupting the calculations by arithmetic underflow exceptions. Before invoking the appropriate code of the program, we perform JVM warm-up to avoid overhead time of JIT compilations. Also, we call Garbage Collector before each test. We run experiments for each particular model 5 times, and the presented results are the average values. Experiments are run at the following configuration: SSD disk, 2.5 GHz Intel Core i7 - 4 cores, 16 GB RAM on Unix based OS. Maximum JVM heap size is 13 GB.

4.4.2 Results

The proposed approximate reasoning method is evaluated on the diagnostic model which is presented in Figure 4.12. On this scheme, the prefix of a node label indicates the component type, *i.e.*, S stands for a server, D for an IoT device, E for edge device, G for global causes, R for a rack. Experiments are run for the following quantity of devices: 20 servers per rack, 3 to 30 racks, 600 devices of 3 different types per server, 1 edge router per 600 devices. Belief Propagation algorithm with a limit to 10 iterations is run on the Bayesian network. This part is implemented with Figaro library, and the result is not presented on the plots, because the evaluation of the model for the first iteration took 2855 s with a maximum memory usage of 6 GB and the offline stage lasted for 130 s. The average quality of the proposed cache-based RCA method is presented in Table 4.1. The results represent different quality metrics of RCA. We can see that the proposed system achieves high precision and true negative rate. Also, the negative predictive value is at the satisfactory level of more than 70%.

Table 4.1: Accuracy of the proposed method evaluating RCA on the model presented in Figure 4.12

Measure	**Value**
Recall	0.57
Precision	0.99
Negative predictive value	0.73
True negative rate	0.99

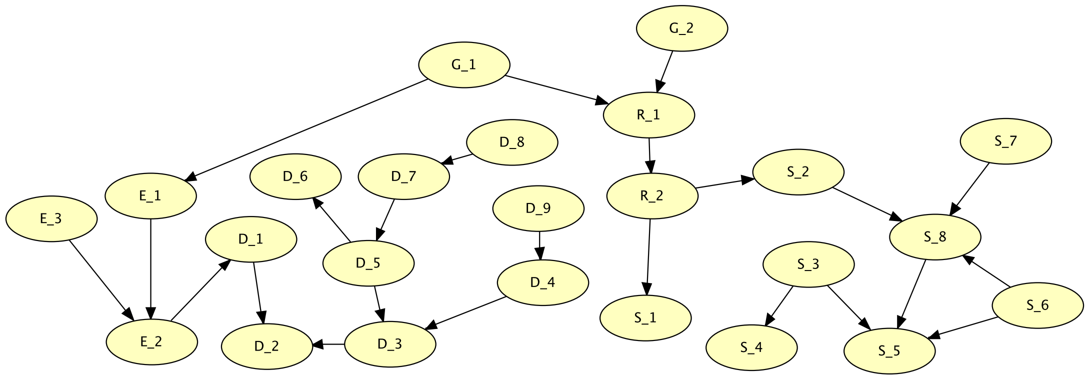


Figure 4.12: Simplified Bayesian network presenting relations between events in different components. One instance of each component type is shown only.

The following plots illustrate maximum memory consumption and time of online and offline stages for the considered algorithms. In Figure 4.13 and Figure 4.14 we present performance results of the offline stage concerning the size of the diagnostic model. The proposed solution is 2 orders of magnitude faster than the compilation of the whole diagnostic model as the one Arithmetic Circuit. Also, the memory used by the proposed solution is much smaller than the conventional AC compilation.

Then, Figure 4.15 and Figure 4.16 present performance results for the online stage (evaluation of models and aggregation of results) with relation to the size of the diagnostic model. The proposed solution is almost 10 times faster during this stage, and it uses 5 times less memory comparing to the evaluation of one AC presenting the whole diagnostic model of a system. There is a trade-off between memory consumption used for cache and memory necessary for the calculations as the system scales. The proposed system with cache uses less memory than a version without cache. The explanation is that storing a single result from the evaluation requires less memory than the memory is necessary for the evaluation of a single compiled model. This feature manifests for networks larger than hundreds of nodes.

In Figure 4.17 we present the evaluation of the maximum memory usage during the reasoning stage depending on the entropy of observations. For instance, the ratio of 60% means that we randomly set 60% of observations on the system variables, and 40% of them is fixed.

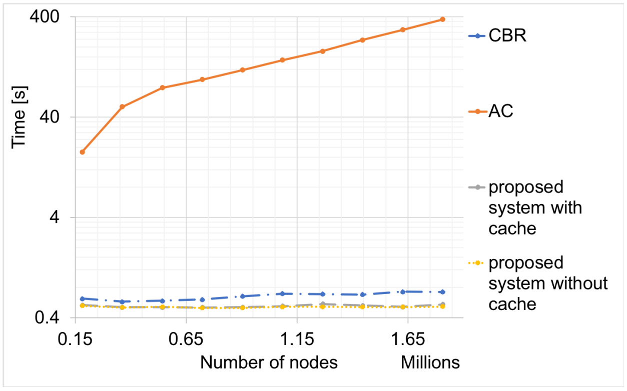


Figure 4.13: Time performance of offline stage (compilation)

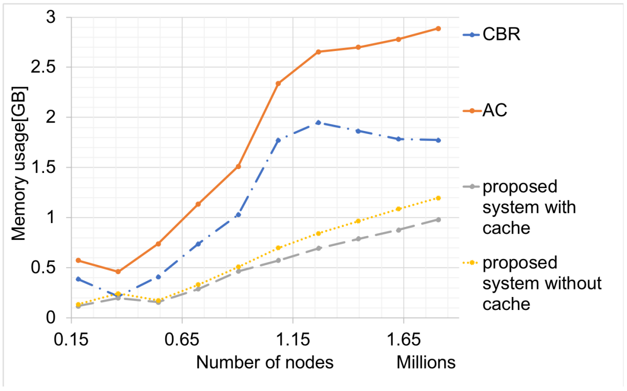


Figure 4.14: Maximum memory usage during offline stage (compilation)

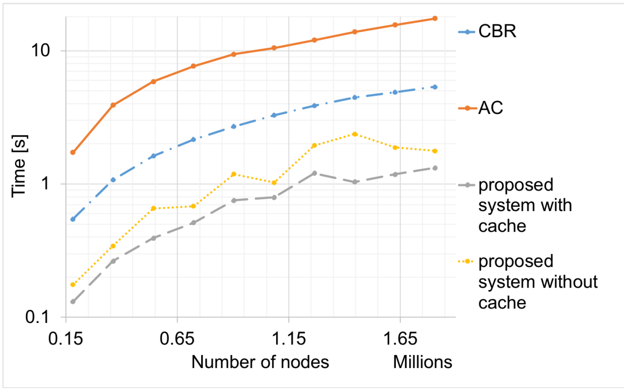


Figure 4.15: Evaluation (online stage) time

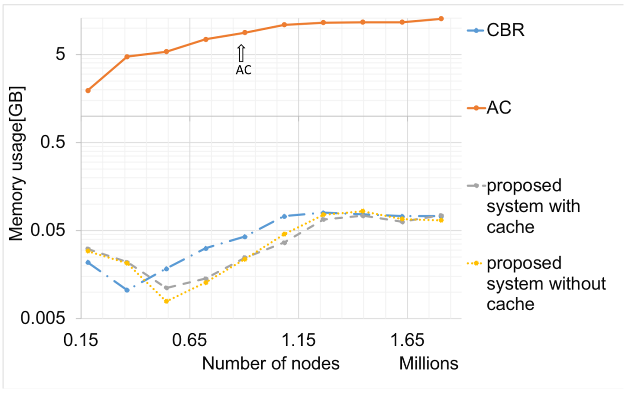


Figure 4.16: Evaluation (online stage) maximum memory usage

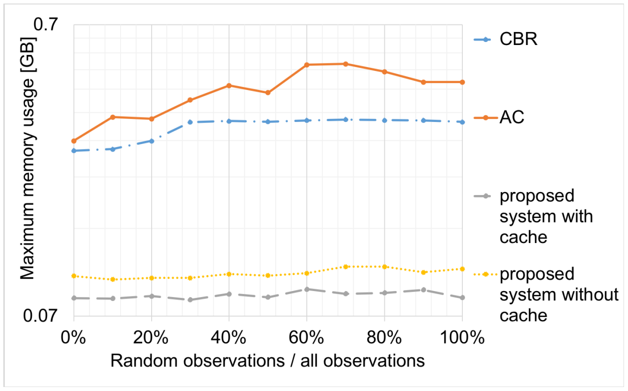


Figure 4.17: Maximum memory usage depending on observations entropy

Summing up, results prove that excellent precision and specificity characterize the proposed method, and acceptable enough sensitivity and negative predictive value. Also, it is about an order of magnitude faster in the evaluation and requires more than two times less memory compared to the accurate approach with compilation the whole BN to a single AC.

4.5 Evaluation: Exploring ability of ABRCA to deal with the dynamism of a diagnosed system

4.5.1 Methodology and experiments

In this section, we evaluate ABRCA. We present our methodology followed by the results of the evaluation. The performance of the algorithm is analyzed in both static and dynamic conditions. The experiment simulates a large IoT and data center environment, and the model is built within the scale and complexity of a real scenario. The example large-scale system which needs ABRCA can be a data center cooperating with an IoT environment, described in Section 3.2. The data are pulled from devices such as sensors, actuators, and smartphones through gateways to the data center to process them in analytics applications.

The actor based root cause analysis is implemented in Scala[1] using Akka[2], which provides an actor model framework, and Ace 3.0[3], which is responsible for Arithmetic Circuits compilation and inference. The presented results are an average of 10 runs of the algorithm in the specified conditions.

Each of the devices in this scenario has defined the diagnostic model in the form of a Bayesian network. For example, compilation, as well as evaluation of a BN having about 20 two-state nodes, utilizes less than 2 MB of memory. The largest evaluated system consists of about 37k devices, and the diagnostic model of the whole system comprises more than 186k nodes in the BN. Centralized RCA with split Arithmetic Circuits running on the same resources is used as a baseline. Results of this part of the experiment are presented in Figure 4.18.

Secondly, we evaluate the dynamics of ABRCA, while devices simultaneously connect to a system during its normal operation. The effectiveness of ABRCA can be explored by measuring the time needed to connect and prepare diagnostic models for the new set of devices being added to the diagnosed system. The results can be seen in Figure 4.19.

4.5.2 Results

We have evaluated the performance of ABRCA to decide if it would cope with the conditions of a real environment. We show that our solution is faster when the more resources are available. It can

[1]https://www.scala-lang.org/
[2]http://akka.io/
[3]http://reasoning.cs.ucla.edu/ace/

be seen in Figure 4.18 comparing performance results obtained from 8 and 12 cores machines. It is observed in the same figure, that the evaluation of ABRCA is 3 times faster than the approach with centralized and non-distributed split Arithmetic Circuits. However, the compilation phase is slower than the one with the centralized approach. The reason is the communication overhead, while some of the devices compile a diagnostic model and some wait for the compilation result to be delivered from another device the same type. In Figure 4.19, we can see the time performance of connecting new devices to the system. The measured time includes exchanges of the compiled models between the devices for different model sizes.

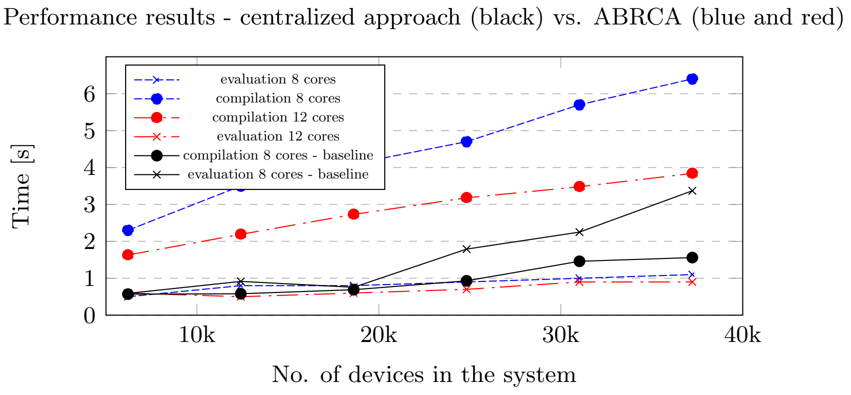


Figure 4.18: Performance of ABRCA - models compilation and evaluation run on different resources

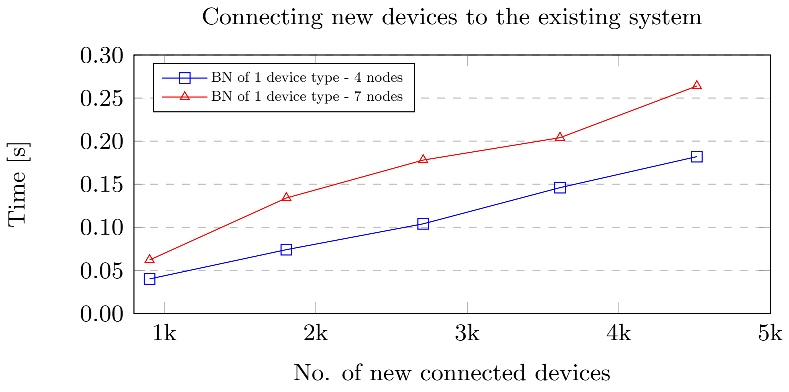


Figure 4.19: Performance of ABRCA - time for system to be ready and perform diagnostics - new devices are connected to the system which initially contains 30k devices

4.6 Discussion, conclusion and future work

In this chapter, we proposed a novel actor based root cause analysis algorithm, which fills the gap between two emerging research areas. These are distributed self-diagnostics solutions available for wireless sensor networks and complex centralized RCA systems used in data centers. The cutting-edge design and abilities of ABRCA are ideal as a core for the future RCA systems working with CPS which consists of smart devices pushing data into the cloud. In the future research, the interesting direction can be the deployment of ABRCA on the real IoT infrastructure, *e.g.*, this provided by IoT-LAB[4]. It will enable to utilize wireless sensors, robots and other intelligent devices which are dispersed in Europe in several laboratories. In this case, each device in the environment will have to handle model compilation, evaluation, and communication with other devices. Then an interesting research branch might be an intelligent system for management of resources delegated to perform ABRCA. Also, an automatic benchmarking system to self-adapt the optimal ratio between devices waiting for the compiled model and compiling one on their own can be considered.

Further research in the area is focused on the deployment of the fast RCA system for efficient diagnostics in Big Data systems for the smart city. Another significant step to take is the creation of a new compilation algorithm for Bayesian networks to leverage repeated structures and improve the accuracy of the method proposed in this thesis. It will be achieved by more complex analysis of the nodes' dependencies between components.

[4]https://www.iot-lab.info/

CHAPTER 5

KNOWLEDGE INTEGRATION AND RE-USABILITY FOR ROOT CAUSE ANALYSIS

Systems and their architecture change very rapidly in response to business and user demand. Many organizations see value in the maintenance and management model of NoOps. The path to NoOps involves not only precise and fast diagnostics but also reusing as much knowledge as possible after the system is reconfigured or changed. The biggest challenge is to leverage knowledge on one IT system and reuse this knowledge for diagnostics of another, different system. We propose a framework of weighted graphs which can transfer knowledge and perform high-quality diagnostics of IT systems. We encode all possible data in a graph representation of a system state and automatically calculate weights of these graphs. Then, thanks to the evaluation of similarity between graphs, we transfer knowledge about failures from one system to another and use it for diagnostics. We successfully evaluate the proposed approach for Spark, Hadoop, Kafka and Cassandra systems.

5.1 Weighted graph framework unifying system information for automatic knowledge transfer

Today's IT systems are large, dynamic, complex, and heterogeneous. The current and the future systems will frequently change their architecture and resources according to the business and user demand. Diagnosing them efficiently in satisfactory time (less than minutes) is already not within reach of even the most experienced operators. Because of that, the majority of trends and efforts around the development of troubleshooting and diagnostics of IT systems is driven by NoOps[1][2][3] business model [98]. NoOps stands for no operations. It means a scenario of fully automated and self-manageable IT infrastructure. The shift of conventional operations to NoOps model is achieved by the full automation of maintenance activities, including failure diagnostics. In this model of maintenance, problems occurring in an IT system are solved immediately without any human intervention.

However, to operate successfully in such a business model, the future diagnostic systems should not only perform precise, automated and fast root cause analysis. These solutions should be able to diagnose problems even in a scenario where there is none or few data about failures and their causes. In many cases, recollecting the data necessary for diagnostics is expensive or even impossible. The use of similar data coming from another system with a different structure is a solution, but it is a considerable challenge. The solutions based on transfer learning can transfer and reuse as much knowledge on the behavior of a system as possible to keep pace with the changing architecture, infrastructure and rapidly growing number of knowledge domains.

So far, we have seen enormous work on automated diagnostics of IT systems, with use of data mining or Artificial Intelligence (AI) [99, 100]. Most of this work uses either metrics or logs for diagnostics. When both are used, the use of logs is limited to counting specific key terms or entries with a specific severity level. Another common limitation of current systems is the lack of inclusion of detailed system information, *i.e.*, connectivity, hardware specification in diagnostics. There is still room for improvement in knowledge integration and knowledge transfer before we reach the era of NoOps. As we show in this chapter, integrating log entries, metrics, and other system data improves the accuracy of the diagnostics for IT systems.

In this chapter, we propose a cross-system root cause classification framework based on similarity

[1]http://cloudcomputing.sys-con.com/node/4054335
[2]https://www.ibm.com/blogs/bluemix/2016/06/moving-devops-noops-microservice-architecture-bluemix
[3]http://www.bmc.com/blogs/itops-devops-and-noops-oh-my

evaluation of weighted graphs with multi-attribute nodes. The framework uses logs, metrics, configuration and connectivity information to represent the state of a system as a graph. Then, the framework evaluates the similarity between an abnormal state and a collection of previously diagnosed states. By finding the most similar graph in the solution space, we can classify the anomaly and provide a root cause. Moreover, we use automatically calculated weights to highlight the system metrics that better describe a failure. Finally, we use the framework for a cross-system failure classification. By acquiring a collection of diagnosed anomalies for one system architecture, we can establish the root cause of anomalies that occur in a completely different architecture (cross-system diagnostics). We leverage the proposed framework for the problem of rapidly changing system architecture as a consequence of changed business or user demands and requirements. Thanks to the knowledge transfer, just after starting a new architecture of a system, we can diagnose it and proactively avoid failures. The proposed system does not only allow for precise diagnostics but also helps in proactive avoidance of failures. The system can output the nearest possible future failure as a result of graph similarity evaluation. Such an approach, saves time, effort and results in performance and reliability advantage over competitors.

We evaluate the proposed framework in the environments running representative and different Big Data applications such as Spark [2] and Hadoop [3]. We inject failures into these environments and evaluate the quality of failures classification, reaching more than 70% of both *f1-score* (the harmonic mean of precision and recall) and accuracy. Then, we perform experiments using different architectures with containers running Cassandra [5] and Kafka [4] systems. We evaluate our cross-system nearest root cause classification when the symptoms of failures are known only for one of these systems. We obtain average *f1-score* 77% with the same level of accuracy.

The remainder of this chapter is divided into six sections. In Section 5.2 we describe the background for the graph similarity calculation. Then we present the framework for creation and similarity evaluation of automatically weighted graphs representing a system's state that contains: metrics, logs, system connectivity, infrastructure. Our contributions are:

- A solution on how to include logs in a graph representation of a system state. (Subsection 5.3.1)

- A method for automatic adjustment of weights of nodes and node attributes, according to the distribution of a metric. (Subsection 5.3.2)

- Evaluation of the proposed solution on real datasets for root cause classification. This Section presents an evaluation of the proposed framework on a cluster running Hadoop and Spark jobs. We prove that including logs and the automatic importance assignment system increases the accuracy of the classification with respect to other methods. (Section 5.4)
- Evaluation of root cause classification in cross-system transfer learning; We search for a failure using knowledge captured from one system (Kafka) and utilize it in another system (Cassandra). We prove that the graph approach can transfer knowledge to/from Cassandra from/to Kafka. (Section 5.5)

Both evaluation sections contain results from total four use cases running in different infrastructures: on-premise cluster and containers in a cloud. This strategy allows us to prove the reproducibility and broad usability of the proposed framework. We conclude the paper with the discussion and plans for future research in Section 5.6.

5.2 Similarity between weighted graphs having multi-attribute nodes

In this section, we provide background knowledge on the problem of similarity calculation between graphs. We define the problem, the graph representation, and how to calculate similarities between different node attribute types.

Definition 7 *We define a multi-attribute weighted graph* $\boldsymbol{G}$ *as an ordered tuple of* (E, V, W, A, S) *that comprises a set of edges* E*; set of vertices* V*; set of weights* W *that are defined for each attribute, vertex, and edge; set of attributes* A *that are defined for edges and nodes; and a set of similarity functions for different attribute types.*

The similarity measure is an intuitive number giving the idea of how two objects are similar to each other. There are many metrics used, usually such measures are the inverse of distance metrics, *e.g.*, $s(x, y) = 1/distance(x, y)$, $s(x, y) = 1 - distance(x, y)$. The greatest the distance in a metric space is, the less similar compared objects are. According to [43], we define the problem of finding a similarity between two graphs as follows.

Definition 8 *For given two graphs* G_1 *and* G_2 *find an algorithm to calculate the similarity* s *of the graphs, which returns the maximum number between 0 and 1. Two graphs have similarity* $s = 1$ *only when they are identical while a similarity value of* 0 *intuitively says that they are completely different.*

Calculating the similarity between the two graphs is a non-obvious problem, especially when the topology of graphs and attributes of nodes are different. Many combinations of node and edge similarity calculation are possible. To find the maximum similarity between the graphs, it is necessary to solve the following optimization problem.

Definition 9 *We define the mapping as injective functions: for nodes –* $m_n : V_1 \to V_2$ *returning the matched node of* G_1 *into* G_2 *and for edges –* $m_e : E_1 \to E_2$ *returning the matched edge of* G_1 *into* G_2

Then we can define the optimization problem.

Definition 10 *The optimization problem is to find the mapping that maximizes the similarity between the two graphs, given by the formula:*

$$\frac{\sum_{v \in V_1} (w(v)+w(m_n(v))) \cdot s(v,m_n(v)) + \sum_{e \in E_1} (w(e)+w(m_e(e))) \cdot s(e,m_e(e))}{\sum_{v \in V_1} w(v) + \sum_{v \in V_2} w(v) + \sum_{e \in E_1} w(e) + \sum_{e \in E_2} w(e)}$$

$$\underset{m_n, m_e}{\arg\max}$$

The above formula states that the similarity is the weighted average of the similarities between the optimally mapped elements of the graph through the mapping functions m_n and m_e. The weights of nodes $(w(v) + w(m_n(v)))$ and edges $(w(e) + w(m_e(e)))$ can help in increasing importance of the nodes and edges that are critical for a particular graph representation.

5.2.1 Approximate Graph Similarity Calculation

Graph representation of a system state. Graphs allow representing an IT system state including all types of data which can describe that state. Each of the system components is a node that has multiple attributes and represents a different level of abstraction, *e.g.*, hardware, server, an application, application module, application thread, container, or a microservice. Edges represent the connectivity between system components. Attributes of a node contain different information encoding the system state, *e.g.*, metric value, log entries, component type, software details.

Also, to represent the different importance of each of the attributes, we introduce **weights** at each level of the graph structure. We use them with each element of a graph: edges, nodes and node attributes. A weight indicates how significant is the influence of the similarity between particular elements on the final similarity result. An expert can define weights through the root cause analysis framework. When an anomaly is detected inside the system state graph, the expert can pinpoint the metrics and components that are more important inside that anomaly. These will be later used as inputs by the root cause classification system. Such an intuitive mechanism creates permanent opportunities for the framework to gather the expert knowledge. In Section 5.3, we introduce the automatic weight calculation mechanism.

Graph similarity calculation. We calculate the maximum similarity $s(G_1, G_2)$ between two graphs $G_1 = (E, V, W, A, S)$ and $G_2 = (E, V, W, A, S)$. We use *hill climbing* [101] to solve the above optimization problem. The similarity between two nodes is calculated by using their attributes, which can be both logs and metrics. In the Subsection 5.2.2 we propose different similarity functions depending on the attribute types - custom functions to compare different elements of a graph.

5.2.2 Similarity between different attribute types

We define similarity functions for numerical, vector, categorical and ontological attributes in Table 5.1. Thanks to the different similarity functions, we manage the calculation of similarities between different attribute types, coming from the two compared graphs.

Table 5.1: Similarity functions used in the proposed framework

Type of attributes	Similarity function
Numerical	$1 - scaled\_distance(a_1, a_2)$
Vector	$cos(a_1, a_2)$; inverse Euclidean distance; Minkowski p distance
Categorical	$1\ if\ a_1 == a_2\ else\ 0$
Ontological	Modified Wu and Palmer [102] similarity metric: $\frac{2 \cdot d(C)}{d(c_1)+d(c_2)}$ c_1 and c_2 – concepts in the ontology, C – their closest common ancestor, $d(c)$ – distance from the root node

Similarity between numerical attributes. This function is used for those metrics that take numerical values such as CPU usage, bytes written to disk or memory used to name a few. More specifically, for numerical type attributes a_1 and a_2, we use the formula $s(a_1, a_2) = 1 - \frac{|a_1 - a_2|}{|max - min|}$. Two points that are close on the scale, will have a higher similarity value. They achieve the maximum similarity (value of *1*) only if they are equal.

Similarity between vectors. Vectors can represent a measurable state of a system module, but can also represent text inside a log file, as we will explain in Subsection 5.3.1. The similarity between vectors is usually defined by the value of cosine between two vectors. Also, other metrics that are based on different distance formulas can be used.

Similarity between types. Graph nodes contain attributes which specify a type. A taxonomy is a tree that represents a hierarchy of concepts in a given domain. In Figure 5.1, we present an example taxonomy. Each node in a graph can contain attributes that define its type inside this taxonomy. A function used for similarity calculation between types, given taxonomy is introduced in Table 5.1.

Similarity between categories. They take values that are names or labels, *e.g.*, the image of a Docker container (*e.g.*, Haproxy, WordPress), disk label, hardware model. According to the categorical values, the similarity is 1 when the values are equal, otherwise 0.

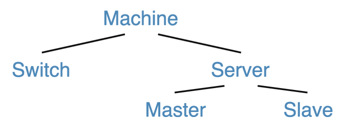


Figure 5.1: An example taxonomy defining equipment type used in the evaluation. For instance, using the ontological similarity formula from Table 5.1: $similarity(Master, Slave) = 0,66$, $similarity(Master, Switch) = 0,4$, $similarity(Server, Switch) = 0,5$

.

5.3 Weighted graphs representing system state for cross-domain diagnostics

Motivated by the challenge of shifting operations to NoOps, we present the following contribution. First of all, we propose a diagnostic framework based on an automatic similarity calculation for graphs representing a system state. The framework automatically adjust graph weights according to the distribution of historical values of metrics. Also, the weight module allows for adjusting the importance of a metric according to an operator's feedback. Weights are used to indicate the important elements of a system which hold significant information for diagnostics. For instance, in case of a network failure, attributes with network metrics and a switch may have more importance and anomalous information than, *e.g.*, CPU load or temperature. The framework reacts to a trigger based on anomaly detection mechanisms, *e.g.*, an error message, exceeded the threshold of a metric. It outputs the similarity score between the current state of the system and previously acquired anomalous states. Such information can be used for early detection of failures and their prevention. In Figure 5.2, we present an automatically weighted graph representing a system state. Blue nodes represent system elements, in this case: hosts, and a switch. Each node contains many attributes which can contain static information, *e.g.*, node type, and runtime data, *e.g.*, metric values, metric distributions. In comparison with state of the art (Section 2.2), in our approach for the cross-system diagnostics, we encode more information. We use attributes of different types in both edges and nodes, together with the information contained in system and application logs, providing a much more detailed input for the graph similarity function. Also, analyzed state of the art shows the dependency between accuracy and the underlying complexity of the solutions. Much state of the art research is focused on accurate analysis and mining of logs based on metadata for a specific log structure. There are not many solutions which diagnose a system just by consuming logs without specific preprocessing techniques. With the solution that we propose, we would like to fill this gap. The solution is as general as possible, and it can work with many IT system types with little human effort to deploy the framework.

In Figure 5.3, we present the proposed framework for root cause classification. The framework manages the creation of weighted graphs and calculation of similarity between them. One graph comes from a repository with anomalous graphs that have been previously labeled with its root cause, and the other one represents an anomalous state of a diagnosed system. Note that we assume the existence of an external anomaly detection system that can extract anomalous system graphs. An expert can label these graphs in the repository, or they can be labeled automatically by an anomaly detector.

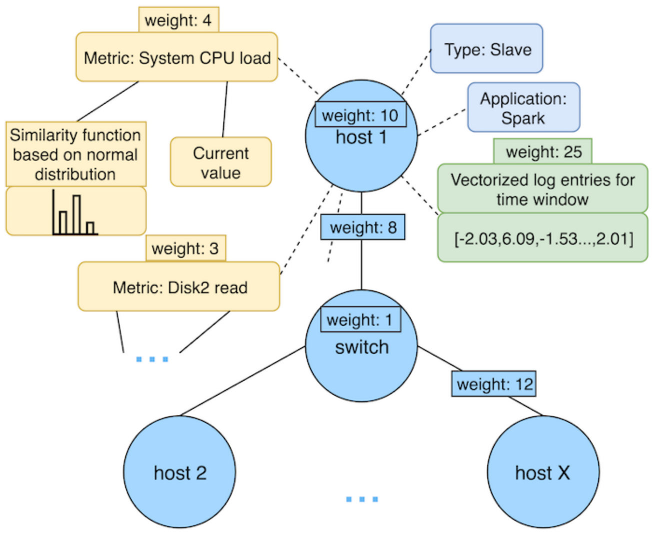


Figure 5.2: An example graph with multi-attribute nodes representing a system state, including connectivity between devices their types, metrics, and logs. Each node contains many attributes, which are different types: categorical, numerical, vector, distribution, classification.

The graph creator builds graphs that represent the system state. They use sources of data coming from different monitoring systems or other information about the system architecture. The content of graphs and their topology depends on the modeling approach. For instance, each node can represent a server, application or its module. The graph similarity module is used to find in a solution space the nearest graph to the anomalous system state graph. By finding this closest labeled graph, we can know the cause of a failure. In case of use of the proposed framework for failure prevention, we get a graph representing the most probable failure which is likely to occur.

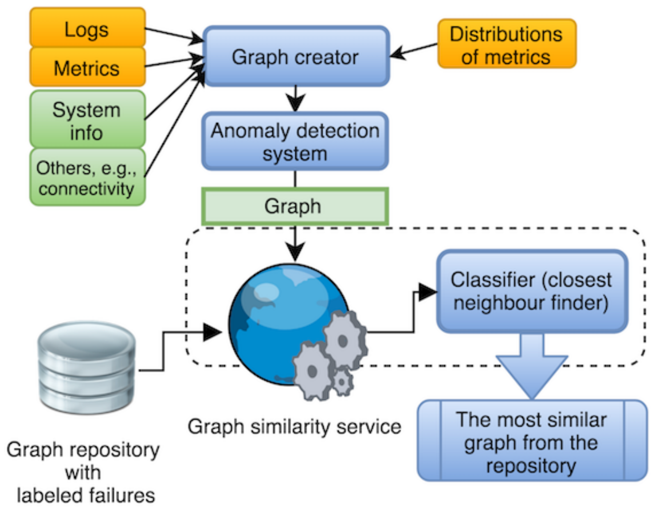

Logs
Metrics
System info
Others, e.g., connectivity
Graph creator
Distributions of metrics
Anomaly detection system
Graph
Classifier (closest neighbour finder)
Graph repository with labeled failures
Graph similarity service
The most similar graph from the repository

Figure 5.3: Scheme presenting the architecture of the root cause classification framework working with an external anomaly detection system.

5.3.1 Including log data

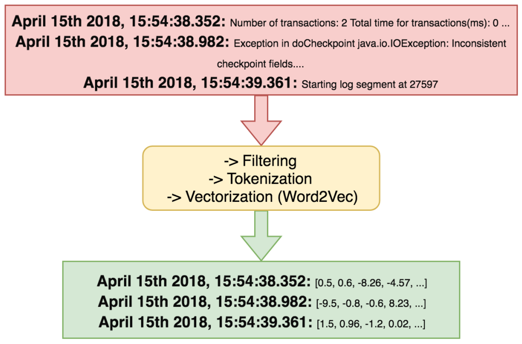
In this subsection, we propose a log representation structure that can be embedded into our graph. In the proposed graph representation of a system state, the attributes capture information from different sources, including logs. In contrast to many state of the art solutions, we consume logs without any metadata or dependency on its structure. Thanks to this approach, our solution is agile and needs a minimal effort for the deployment. We only extract timestamp, severity level, and the rest is treated as a log entry that includes application module name, message, thread name, and other fields. Moreover, users (framework operators) can disassemble logs by modules and put them inside new nodes or attributes representing these modules in a system state graph. For instance, an operator deploying the proposed framework may decide that the graph representation of a system should be a detailed one. Then, a node presenting a host is connected with its child nodes, representing some modules, *e.g.*, threads, classes, application modules. Logs of this host are split among these nodes.

We propose to use vectorized logs using Word2Vec models, in a system's state representation for the following reasons. The whole log processing is a simple algorithm and includes removal of special chars, sequences, and stop words, tokenization, and vectorization. The scheme illustrating the whole process is presented in Figure 5.4.

Filtering. After eliminating special char sequences *e.g.*, hex strings, the vocabulary in logs is limited. Typically, human-created templates of logs do not contain synonyms, just strict and simple phrases. After this stage, log entries contain less noise and represent a state of the generalized system, rather than a particular case. Also, removing special characters helps to avoid model over-fitting. This step does not only improve the model quality but also transforms a log into a universal form, which is mandatory in cross-system diagnostics.

Tokenization. The tokenization step disassembles sentences into bags of words.

Vectorization. Thanks to Word2Vec we transform log into vectors. The vectorization stage enables to represent log entries in relatively small models, what we show later in the evaluation in Section 5.4. Firstly, it is necessary to create a model mapping the vocabulary into n dimensional space. The performance of a model depends on its configuration parameters and the size of the vocabulary used for training. A considerable advantage of using a Word2Vec embedding model is that it performs well even if it is trained using the vocabulary of one domain and used for another one. Also, similarity calculations should be as fast as possible to enable diagnostics of failures in a dynamic environment. Hence, it is not feasible to use natural language processing (NLP) techniques such as key terms extraction using rank algorithms for each log sentence as we demonstrate in Section 5.4, where we test different approaches. The proposed log processing algorithm does not need much configuration work. We only need to adjust a time window size, which starts with a specific severity type. In this chapter, we propose to use severities with a higher level than the warning one.



```
April 15th 2018, 15:54:38.352: Number of transactions: 2 Total time for transactions(ms): 0 ...
April 15th 2018, 15:54:38.982: Exception in doCheckpoint java.io.IOException: Inconsistent
checkpoint fields....
April 15th 2018, 15:54:39.361: Starting log segment at 27597
```

```
April 15th 2018, 15:54:38.352: [0.5, 0.6, -8.26, -4.57, ...]
April 15th 2018, 15:54:38.982: [-9.5, -0.8, -0.6, 8.23, ...]
April 15th 2018, 15:54:39.361: [1.5, 0.96, -1.2, 0.02, ...]
```

Figure 5.4: An example process of transformation log entries to vectors using Word2Vec. After this transformation, each log entry is represented (embedded) by a vector in a continuous model space. A vector of a log entry is computed as the average vector from all vectors representing words in this entry.

After a failure occurs, we can find messages on the logs containing information for that failure, while some others are just messages belonging to the usual operation of the system components. As discussed in [103], using smaller time windows capture the more detailed meaning of a word (in our case, if it mentions a failure), and large ones which capture the context (general context of the application used). We propose to use two log windows: one called (1) **context window**, and the other one (2) **event meaning window**. The context window represents the general state of a part of the system. Mainly, it enables to capture application's normal activities. The event meaning window captures log entries in a shorter time after a particular event. Logs in such a window represent specific information about the event. Both windows start when an error or warning message is written into the log. The reason we take this approach is that operators usually do not know when the system starts failing, but they know the precise time of every error or warning written to logs. We explain the concept of window lengths in Figure 5.5.

5.3.2 Using metrics distribution for automatic weighting of node attributes and measuring similarity

The distribution of values for a metric can be used to know how different or uncommon is a value observed in the system. In this subsection, we explain how we use the cumulative distribution function

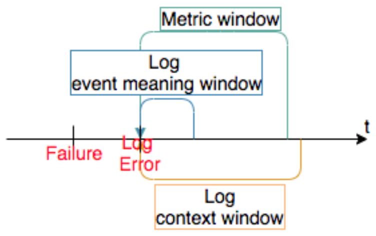


Figure 5.5: Scheme presenting context window and event meaning window used for extraction of logs. Metric window is used for extraction of metrics. Both windows start on a first Error or Warning message.

to our advantage by, firstly, calculating weights automatically inside the graph representation, and secondly, comparing two numerical attributes considering the distribution of their historical values.

Automatic weighting of node attributes

There are two ways of defining weights in graphs which represent the importance of the different elements of the system status.

Firstly, thanks to the weight assignment mechanism, operators adjust the importance of a particular metric in the graph representation based on their expert knowledge of a failure. For instance, operators might put a higher weight on the CPU load than on the disk I/O, for a problem related to a system overload. Thanks to this approach, we do not require operators to know specific characteristics or deviations of metrics. We use a part of their expertise which contains importance of metrics used in a troubleshooting process.

The second possibility for weight assignment in graphs is an automatic weight calculation from available metric data.

In this subsection, we focus on the latter. We propose an **automatic weight assignment** mechanism to automatically assign the importance of an attribute, given its distribution.

According to the troubleshooting activities of IT operators, the more abnormal the attribute value is the better describer of a particular failure. In this case, we define weights which are proportional to the deviation of the usual value for an attribute distribution of values. Again, using the normal distribution $X \sim \mathcal{N}(\mu, \sigma^2)$, we have the following definition.

Definition 11 *The weight of a numerical attribute which is proportional to the deviation of a metric value a is defined as* $w(a) = \frac{|a-\mu|}{\sigma}$

Measuring similarity from metric distributions

The similarity function based on metric distribution enables to utilize data containing historical values for an attribute. The function definition contains cumulative distribution function (CDF) and its parameters. We define the similarity function between two numerical attributes, as the formula $similarity = 1 - distance$ where $distance$ is the difference between CDF values of attributes. For the normal distribution used in the proposed framework, we have the following definition.

Definition 12 *Given numerical attributes a_1, a_2 from two graphs and distribution of these attributes $X \sim \mathcal{N}(\mu, \sigma^2)$, their similarity is provided with the formula $similarity = 1 - |\phi_{\mu,\sigma^2}(a_1) - \phi_{\mu,\sigma^2}(a_2)|$*

The above two simple mechanisms allow to automatically include the importance of attributes in the graph representation of a system state and similarity calculation.

5.3.3 Enabling cross-system diagnostics

Finally, we use the proposed framework to transfer knowledge about failures from one system that we call *source* system to another that is called *target* system. In Figure 5.6, we present the cross-system knowledge transfer problem. A source system and a target system can have both different topologies and contents of nodes. We use the proposed graph representation of system states as a medium to transfer knowledge about failures. Then, thanks to the framework, we can compare two states of different systems and calculate the maximum similarity of these states. In the final step, we find the nearest graph, which best describes a target system state by knowledge coming from a source system.

In details, using our framework, knowledge transfer is possible because of:

1. Calculation of the maximum similarity between two graphs with different structures using different similarity functions (Subsection 5.2.2). The framework finds the maximum similarity by matching proper subgraphs. Also, defining a taxonomy allows for the calculation of the similarity between two nodes that are different but represent the same concept in a domain. For instance, a slave server of Spark and a data node of Hadoop are close to each other inside the taxonomy, because they are both slaves in a master-slave architecture.

2. Inclusion of logs in the graph representation, as they describe in natural language events that happen in the system, independently of their architecture or resource usage (Subsection 5.3.1).

The two log windows (context and event) contain universal descriptive information, no matter what the differences are between the topologies and components of the two system graphs.

3. Including the information contained in the distribution of the metrics for a given architecture. We do it through the automatic weight assignment and the similarity function based on the distance between distributions (Subsection 5.3.2). The metric values registered for the source and target system can be very different depending on their resource usage patterns. Calculating weights and measuring the similarity using their distributions, allows for a better comparison between two different systems.

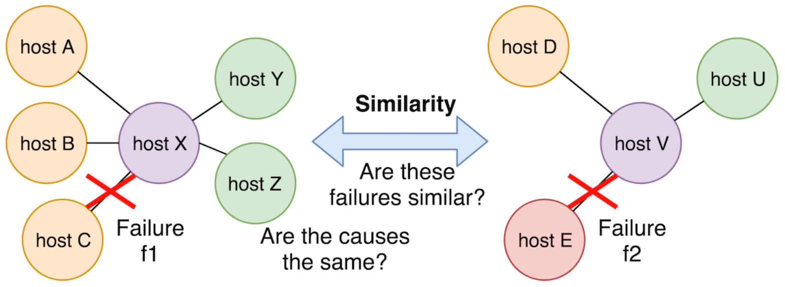


Figure 5.6: Scheme illustrating an idea of cross-system graph comparison

5.4 Evaluation: Root cause classification through finding the nearest graph

In this Section, we show through a series of experiments the quality of our proposed root cause classification framework. We evaluate different features of the framework and compare them to representative and popular state of the art techniques. We use a *f1-score* metric which both includes recall and precision. In this Section, we evaluate the quality of the framework in a scenario where the source and the target system is the same. For this task, we use two use cases: a Spark cluster and a Hadoop cluster. We evaluate cross-system diagnostics in Section 5.5, using Kafka and Cassandra systems.

5.4.1 Experimental methodology

Experimental environment. In the first set of experiments, we use an on-premise cluster described in Section 3.3.1. The monitoring system acquires 22 metrics representing the system state, such as CPU total load: `idle, iowait, softirq, system, user`; disk: `bytes read, bytes write, IO read, IO write`; memory: `buffer cache, free, map, used`; network: `received bytes`, `received packets, send bytes, send packets`, and processes: `load10, load15, load5, number of running processes`. The power meters acquire energy consumption of the servers and the switch. The probing period is set to 5 seconds. The monitoring system works on InfluxDB[4] stack and we use ElasticSearch[5] stack for log storage.

Workloads. During the experiments, we generate Hadoop and Spark workloads using HiBench [104]. We use workloads such as sort, word count, k-means clustering, Bayesian classifier. Each workload takes from 20 min to 2h. Random workloads run continuously.

5.4.2 Types of injected failures and anomalies

We inject different failures in the experimental environment. Each of the described failures is injected 20 times. We choose a set of failure types which are representative and well-aligned with use cases in real environments. Also, different failures should manifest exclusive symptoms in different metrics and logs. The next criterion of choosing the failure types is that they should differentiate possible scenarios of lacking data that are often caused by connectivity problems.

The following list presents the injected anomalous workloads and failures.

[4]https://www.influxdata.com/

[5]https://www.elastic.co/

- **High CPU load**. Background process running CPU pattern of 100% load for 90% of server cores. This failure simulates a scenario of a node slow-down, caused by *e.g.*, an unfinished job, unwanted or unfinished process. CPU performance degradation can also simulate a failure of one of many workers in a Big Data cluster.

- **High disk load**. Random write and read operations on a 10 GB file, generated with the FIO utility[6]. This failure simulates a scenario of a failed disk in a disk array. Thanks to this failure type we can observe many HDFS errors.

- **High network transfer**. 20 threads are uploading and downloading 5 GB files. It simulates significant network slowdowns, which can occur as a result of network infrastructure failure.

- **Host shutdown**. Immediate node shutdown through IPMI card. It simulates a node crash, a sudden and unexpected failure of the whole machine.

- **Network failure**. Physical disconnection.

The symptoms of failures have an understandable impact on system metric values. As we mentioned before, we include power metrics of the servers and the switch. Regarding the switch power, we can observe different peaks and power values depending not only on the network transfer but also on the connection and disconnections. In Figure 5.7 we present the switch power distribution depending on the injected failure, and the referential distribution for the system running random workload without any failure injected. We can observe that different power consumption values characterize different failures. These distributions increase the quality of failure classification in similarity evaluation. For instance, high disk load manifests in a low switch power consumption, while high network use manifests in significantly higher median value.

To evaluate the quality of the root cause classification, we use *f1-score* metric that is defined as follows.

Definition 13 *Let TP stand for number of true positives in multi-class classification task, then FP stands for false positives. We define recall (true positive rate) as* $\frac{TP}{P}$*, and precision as* $\frac{TP}{TP+FP}$*. Then* ***f1-score*** *is the harmonic mean of precision and recall:*

$$f_1 = \frac{2}{\frac{1}{recall} + \frac{1}{precision}} = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

[6]https://github.com/axboe/fio/

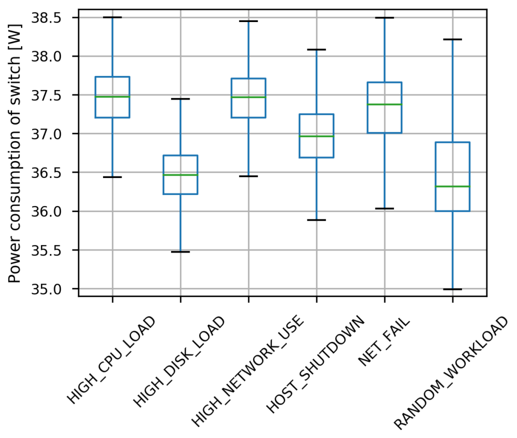


Figure 5.7: An impact of different failure types on the power consumption of a switch. In random workload no failures are injected. Intuitively, average power consumption for *net failure* is one of the highest because of connection and disconnection events, while for *high network use* switch has to handle the abnormal traffic. Most probably, influence of *high CPU usage* on switch power consumption can be explained by higher number of connections generated by a node, one some processes are blocked.

5.4.3 Evaluation: Leveraging logs for root cause classification

We evaluate different methodologies and their configurations for the use of logs for the classification task. In the evaluation, we present the result of solving the following problems.

- **Model training vocabulary.** We fit Word2Vec models using different vocabulary. It can be a specific vocabulary for a particular domain or a general dictionary *e.g.*, English one. For instance, we can train such a model with logs from Spark cluster and use this model to vectorize Hadoop logs.
- **Model size.** We evaluate different numbers of dimensions of a vocabulary space (vector size).
- **Key terms extraction.** We compare the performance of the use of the whole available log entries with the key terms describing the system state.
- **Log window length.** Size of the window is a trade-off between generalization of logs and capturing precise event information. Taking to much text can fuzzify the meaning of the event, and opposite, taking too little text can mangle an analyzed system state. We evaluate different window lengths for both event and context windows.

We create Word2Vec models with the process described in Subsection 5.3.1 evaluating different vector size and vocabulary used for model training. In Figure 5.8, we see average *f1-scores* of the failure classification for the two use cases: the Spark and Hadoop cluster. We present only the best

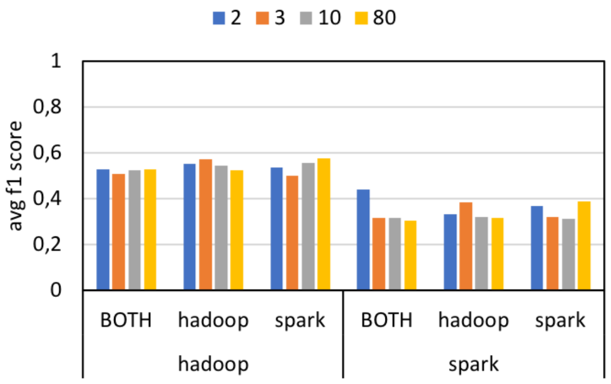


Figure 5.8: Plot presenting the quality of root cause classification depending on the number of dimensions used in Word2Vec model, and the training vocabulary source. Log window length: 30 s

results achieved during the evaluation of different log window sizes. Also, we present summarized results of vector size evaluation. For vector sizes between 3 and 80 *f1-scores* does not change much. In the Figure 5.8, the inner groups stand for the source of the vocabulary used for model training. As well as for Hadoop and Spark, the classification performs the best when the same vocabulary is used for model training and vectorization. For both use cases, the models perform well with small vector sizes - 3 for Hadoop and 2 for Spark.

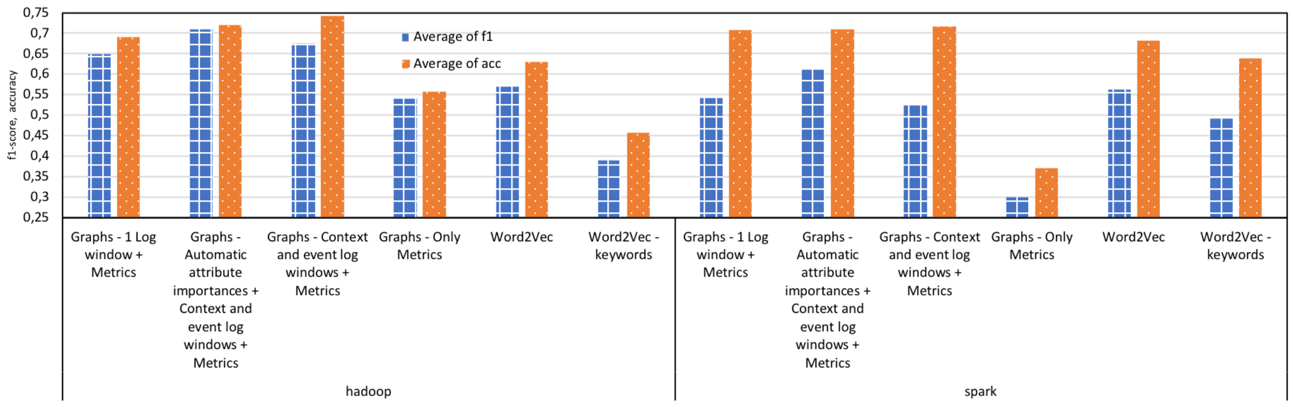


Figure 5.9: Plot presenting root cause classification quality depending on the mechanism used. Average *f1-score* is calculated from all of the injected failures. The proposed framework performs better than state of the art solutions (Word2Vec).

In the next step, we test different approaches of extracting information from logs and representing it in graphs. For the first approach, we use Word2Vec, as described above. In the second approach, we use SGRank [105] algorithm to extract key terms which best describe a system state. This algorithm combines statistical methods, *e.g.*, TF-IDF, with graph-based approaches of key terms. In Figure 5.9, we confirm that using the whole text is the best method to represent the log meaning [80].

5.4.4 Evaluation: Root cause classification via similarity of weighted graphs

In this subsection, we present the results of evaluation of the root cause classification. We test four different configurations of the proposed framework and compare them with the state of the art methods. We show how augmenting the dataset used for the classification task improves its performance. In Figure 5.9, we present the results of the evaluation: average *f1-score* and accuracy. Average *f1-score* is calculated over all of the injected failures. In evaluations where it is emphasized that we use automatic attribute importance assignment, we utilize both similarity function based on distribution and automatic weight calculation. In others, we use equal weights in a graph.

We can see that the proposed framework that contains context and event log window and automatic attribute importance calculation performs better than state of the art methods. Considering performance for two use cases, graphs with automatic weights reveals the best performance. Regarding the Hadoop use case, accuracy reaches 0.72, and *f1-score* reaches 0.71. As for the Spark use case, *f1-score* is a little bit lower 0.61 and accuracy 0.71.

We evaluate the proposed framework for different event and context window lengths. In Figure 5.10, we present detailed results of this evaluation. The performance changes smoothly, there are local maxima of f1-score. These maxima show balance points between log generalization and extraction of precise information about a particular event. The greater is log window length, the more fuzzified information about an event is held in the analyzed window. Note that for Spark, there is also local minimum of 0.51 for event window length of about 10 s and context window length of 60 s. Most probable explanation of this minimum is the timeout for idle executor which is set by default to 60 s.

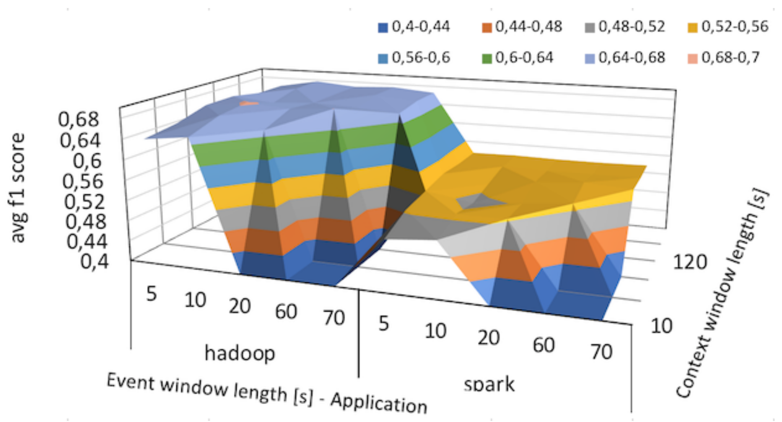


Figure 5.10: Plot presenting quality of failure classification via graphs with equal weights depending on the log window sizes. Average *f1-score* is calculated from all of the injected failures.

In Figure 5.11 and Figure 5.12 we present detailed evaluation results for each of the injected failures. We compare the use of logs with the proposed framework comprising automatically weighted graphs. The proposed framework performs significantly better than Word2Vec, especially with the classification of *high CPU load* and *host shutdown*. There is no observable difference in the performance of the proposed framework when used for the Spark or Hadoop use case. The exception is *high network transfer*, which is classified well only for Hadoop by both Word2Vec and the proposed framework. *High network transfer* manifests in characteristic log entries for Hadoop, and for Spark only in network metrics. Also, it is important to emphasize that, received results come from similarity evaluation of graphs created automatically without any weight adjustment by a human.

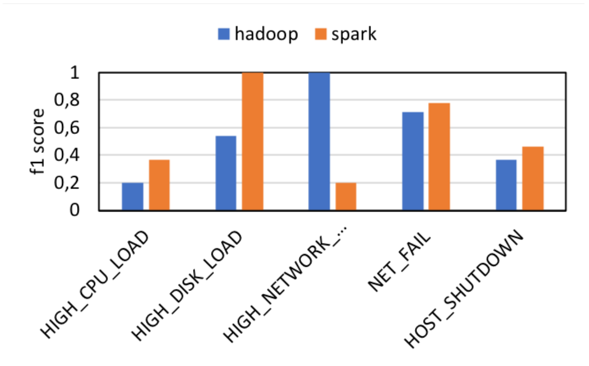


Figure 5.11: Word2Vec model with parameters reaching the maximum quality, chosen from Figure 5.8. Log window length: 30 s

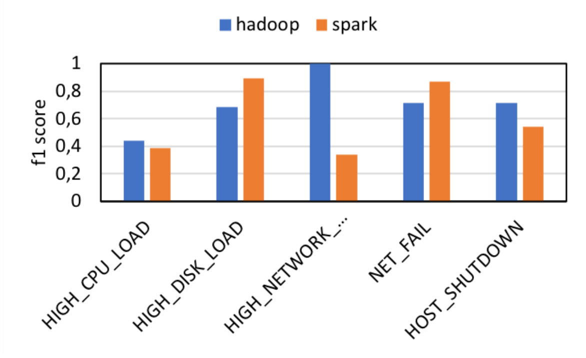


Figure 5.12: Automatically weighted graphs. Context window length: 30 s, event window length: 10 s, metrics window length: 120 s

5.5 Evaluation: Cross-system diagnostics - transferring knowledge

In this section, we evaluate our approach in a more cloud-oriented environment, by running microservice architectures made up of containers.

5.5.1 Experimental environment

We use Grid'5000 a customizable testbed that provides access to different computing resources and infrastructures. We deploy a cluster of 7 virtual machines with 16 GB of RAM and four cores. We install DC/OS[7] on these machines, a container orchestration tool that will allow us to deploy the microservice architectures. The setup is 1 master node, 1 public node, and 5 private nodes. Additional information about DC/OS parts can be found in their website[8]. We use two additional representative Big Data architectures to perform root cause analysis with them. The first one is a Cassandra deployment with 5 Cassandra containers that are going to be continuously queried by 10 containers with Yahoo Cloud Service Benchmark [106] installed. The second one is a Kafka architecture, in which we have 5 brokers, 10 producers that push messages to the Kafka cluster and 10 consumers that read those messages. Additionally, the Kafka brokers need a Zookeeper [107] instance to coordinate them. A simplified version of the graph representations we use for these deployments is shown in Figure 5.13. Note that these two architectures are very similar with a decentralized cluster of servers or brokers that interact with each other and clients that read or write data into this cluster. This scenario is a suitable one for our knowledge transfer approach since failures that happen in one system will have a similar effect if they also occur in the other one.

5.5.2 Methodology

Regarding the failures, we injected them in both the hosts and the containers. For the hosts, we use the same high CPU, high disk, and high network transfer anomalies as in the Spark scenario to stress the machines. For the containers, we pause them through `docker pause` instead of using host shutdown and network failure. We do so because a container cannot be physically disconnected from the network as a host would. The anomalies are injected six times each, in one random element of the architecture for 120 seconds.

[7]https://dcos.io/
[8]https://mesosphere.com/

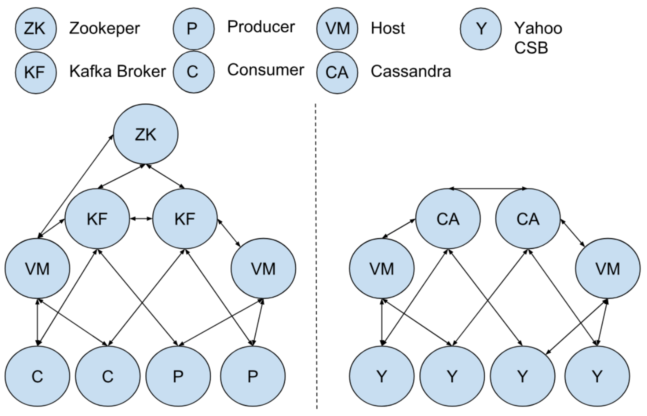


Figure 5.13: A simplified version of the graph representations we use for the microservice architectures. On the left the Kafka architecture with a Zookeeper instance coordinating the brokers and producers and consumers using the message queue. On the right a Cassandra cluster with the YCSB clients. Notice how the VM's are connected to the containers they are hosting through edges that represent this relationship.

5.5.3 Evaluation: Cross-system diagnostics

We present detailed results of the evaluation in Figure 5.14. Average *f1-score* is 0.77 in case of using Cassandra as a source system and Kafka as a target one. In the reversed configuration, the result is 0.76. Note that the scores of the cross-system diagnostics are better than the first evaluation of the framework, due to the different number of types of the injected failures. Both quality results are approximately equal, thanks to the symmetry of similarity function. The small difference is caused by the task of finding the nearest graph (a one with the highest similarity number). This operation is not always symmetric. Considering that two systems are different, in their topology, behavior, and logs, the results are showing high performance of the proposed framework.

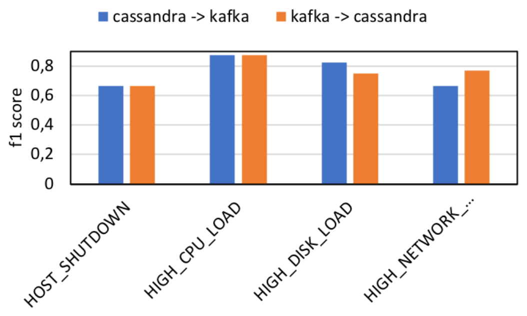


Figure 5.14: Plot presenting results of cross-system diagnostics via finding the nearest graph representing an anomalous state of a system. Results of two cases are presented. 1) *Source system*: Cassandra, *target system*: Kafka; 2) *Source system*: Kafka, *target system*: Cassandra. Average *f1-score* and accuracy: 1) 0.76, 0.77; 2) 0.77, 0.77.

5.6 Discussion, conclusion and future work

In this chapter, we proposed a framework for finding the nearest failure cause via similarity evaluation of weighted graphs. The framework is aimed to diagnose one system when the knowledge about failures is acquired from another system with a different structure. It can mean a use case where a new system has just started operating, it fails, and it is hard to diagnose it. Also, the proposed framework aims to facilitate knowledge transfer between systems and operators. Firstly, we described the whole framework and its contributions. The most significant contributions are automatic calculations of metric weights; integration of logs with system topology and metrics into graph representation of a system; and leveraging historical metric values for similarity calculations. Then, we evaluated the proposed framework in total with four different systems. We inject common anomalies and failures, such as hardware overload, node crash, and network disconnections. In the first evaluation section, we use Spark and Hadoop clusters. We confirm the quality of root cause classification that achieves average *f1-score* of 0.71 for Hadoop and 0.61 for Spark. These results show that the framework outperforms state of the art methods. In the second evaluation, we utilize a cloud environment of containers. We evaluate cross-system diagnostics via knowledge transfer - diagnosing a target system when knowledge about failure causes and anomalous states is known only from a source system. We run a scenario of Kafka acting as a source system, Cassandra as a target one, and a reversed one. Cross-system diagnostics reaches *f1-score* of 0.77. The achieved results confirm that the proposed framework, and in particular its ability of knowledge transfer, allows reaching the state of self-manageable IT systems.

In the next stage of research on cross-system diagnostics we focus on:

- Evaluation of the framework on the real large-scale environments. We would like to integrate the framework with a failure prevention system.
- Aspect of explainable knowledge transfer in cross-system diagnostics.
- Distinguishing random errors, and the ones which are critical for the future system performance and reliability.
- Mechanism for automatic propagation of weights for anomalous regions inside graphs.
- Research in the field of predicting failures with the use of transfer learning.

CHAPTER 6

INCREASING PERFORMANCE THROUGH PREDICTION AND PREVENTION OF FUTURE FAILURES

Failed jobs in a supercomputer cause not only waste in CPU time or energy consumption but also decrease work efficiency of users. Mining data collected during the operation of data centers helps to find patterns explaining failures and can be used to predict them. Automating system reactions, *e.g.*, early termination of jobs, when software failures are predicted does not only increase availability and reduce operating cost, but it also frees administrators' and users' time. In this chapter, we explore a unique dataset containing the topology, operation metrics, and job scheduler history from the petascale Mistral supercomputer. We extract the most relevant system features deciding on the final state of a job through decision trees. Then, we successfully train a neural network to predict job evolution based on power time series of nodes. Finally, we evaluate the effect on CPU time saving for static and dynamic job termination policies.

6.1 Different prediction approaches leading to prevention of failures

Data centers are a core element in most IT systems, hosting cloud applications, enabling HPC or performing intensive Big Data analytics. Although the optimal architecture of a data center may be different for each of these applications, general maintenance problems remain the same. Failures in hardware and infrastructure can both cause software failures or may be the result of such software failures. Software errors are the most common cause of failures [57]. Also, many jobs produce large network and storage system loads which degrade the system performance [108].

Data presenting the state of a system is usually so complex that administrators might not take the best decision to recover a system efficiently. Moreover, in many cloud-oriented services, system monitoring information is limited to hardware metrics, and do not include user application logs. Thus, it is even more challenging to predict job failures and take proper action. Evaluating jobs in run-time augments administrative metrics and increases the confidence of taken decisions. Therefore, jobs which are likely to fail or decrease the performance of a system can be terminated in advance. Such an early termination allows to save resources, computing and human time, and it lowers operational costs. According to the dataset used in this chapter, completed jobs in the petascale Mistral[1] supercomputer consume about 45 million CPU hours per month and they are 91.3% of all submitted jobs. Predicting the final job state at the time of job submission and during run-time allows for forcing job termination before a failure occurs, enabling savings. However, deciding when it is necessary to terminate a job is a nontrivial task.

In this chapter, we show insights and results of operational data analysis from petascale supercomputer Mistral. We explore predictability of a supercomputing environment, utilizing this particular use case. Data sources include hardware monitoring data, job scheduler history, topology, and hardware information. We explore job state sequences, spatial distribution, and electric power patterns. We augment datasets during the exploration to show how knowledge coming from job scheduler, monitoring system, and topology and structure, can increase prediction capabilities and uncover new patterns. We discriminate among job submission features that explain the termination status of jobs based on job traces.

Then, we analyze the impact of both **static** and **dynamic** job termination **policies** using different data center metrics. We propose new job state prediction algorithms based on Decision Trees (DT)

[1]https://www.top500.org/system/178567

and Convolutional Neural Networks (CNN). We use power series of nodes to build a model used for failure prediction at run-time. The trained CNN achieves 85% of precision in the classification of failed jobs by power series. The CNN predicts failures for more than 40% of failed jobs in the 20^{th} percentile of their execution time.

We describe used data set in Section 6.2. We show results of above mentioned analysis in Section 6.3. Then we focus on prevention of failures. Section 6.4 presents the extraction of important features and their discovery by means of DTs that are created using these data. Then, in Section 6.5, we describe the training and use of a CNN for job state prediction. At the end of Section 6.5, we show savings applying different policies for early job termination. We discuss results, the usefulness of the proposed policies and include plans for future research in Section 6.6.

Our contributions are:

- Data mining and advanced analysis of data sets describing runtime of the petascale Mistral supercomputer. (Section 6.4)
- Failure prevention policies: static and dynamic one. Static policy is based on Decision Trees and data known at the time of a job submission, while dynamic one uses a Convolutional Neural Network and power of nodes allocated for a job. (Section 6.5)

6.2 Mistral Supercomputer Dataset

6.2.1 Job scheduler history

Through analysis of historical data from the scheduler, we investigate which features are important, thus deciding on a final job state. This goal motivates our strategy, which is oriented to jobs rather than nodes. We use states from the scheduler to determine an output of a job. In the dataset, each job finishes with one of the following states, defined by Slurm documentation.

- **Cancelled** – A user or administrator cancelled a job. The job may or may not have been initiated. In the following analysis, we consider only cancelled jobs longer than 0 s.
- **Completed** – Job has terminated all processes on all nodes with an exit code of zero.
- **Failed** – Job terminated with non-zero exit code or another failure condition. According to Mistral, another failure condition includes failures caused by any external factor to an allocated node, *e.g.*, failures of Lustre FS, IB.
- **Node fail** – Job terminated due to a failure of one or more allocated nodes. This state includes only hardware related problems of a computational node.
- **Timeout** – Job terminated upon reaching its time limit.

Each job consists of one or more steps. A job submission script defines the execution order of steps; also, the order can be read from Slurm history. The order can be sequential, parallel, or mixed, see example script in Listing 6.1.

Listing 6.1: Example Slurm batch script. Two steps run sequentially on 80 nodes.

```
#SBATCH --nodes 80
#SBATCH --tasks-per-node 10
# First step
srun --nodes 80 --tasks 10 mkdir /home/$USER/$SLURM_JOBID
# Second step
srun app.mpi in.csv out.csv
```

Most steps in Mistral dataset are executed sequentially. In the Slurm database, there are 76 columns. They contain information about jobs: (1) job configuration specified by a user, and (2) statistics known at the end of a job. We give more details about these data in Subsection 6.2.3. In this chapter, we consider all above **job** states. For **steps**, the dataset includes: Completed, Failed and Cancelled.

6.2.2 Time series data analysis

Mistral metrics are acquired every 60 s into an Open Time Series Database (OTSDB) instance that is installed on the top of HBase cluster. For this research, the data from the cluster are exported using the HBase ExportSnapshot tool. Then, we import a snapshot with the size of 0.5 TB from a regular continuous period of 10 months of system executions to our analysis environment containing 8 machines with 120 physical cores, 672 GB of RAM. We use Apache Spark for data processing. For training of a CNN, we need job scheduler data merged with power metrics. We merge Slurm steps with data from OTSDB representing power metrics of nodes used by a step during its run-time. That merged steps should contain at least two power measurements. In the worst case, for steps shorter than 120 s, it is possible to merge only one timestamp with node power metrics. So, in the evaluation, we consider a subset of steps longer than 120 s. Discarding short jobs, we do not lose many data: about 1.2M of all steps from the set run for more than 60 s and 1.1M more than 120 s.

6.2.3 Dataset split

We show how different knowledge sources: software – job scheduler, hardware – monitoring system, and platform – topology and structure, impact prediction and classification accuracy. Also, we detect which part of the data increases the prediction capabilities of a model when the only used information is the one known at the time of a job submission; and which part of the data improves classification capabilities, when we use statistics of finished jobs. Datasets are divided into the following sets - named with a capital letter for later reference:

- **Slurm job configuration data**: information of either jobs or steps, which is known at the time of submission e.g., reserved time, allocated nodes, required CPU frequency, start time.

  [we call it ***dataset C*** in the experiments]

- **Slurm user data**: columns with information about prior user allocations. Also, this dataset contains aggregated user data. The set includes factors of jobs terminated with each of 5 possible states to a number of all submissions in different windows. We aggregate the data by user and windows with different sizes: last N submissions (N=1, 100, 1k, 10k). [***dataset U***]

- **Slurm job summary data**: information is known at the **end of a job**, *e.g.*, duration, disk read/write (R/W) – the sum of local storage and Lustre operations done by a job, virtual memory (VM) size, other hardware usage. [***dataset S***]

- **Power metrics of nodes (OTSDB data)**: power metrics of computing nodes (blades). [***dataset P***]

- **Data center topology**: topology and localization of nodes. [***dataset T***]

- **Hardware profiles of nodes**: types of nodes, number and types of CPUs, amount of RAM. [***dataset H***]

In Figure 6.1, a high-level scheme of data processing modules and data sources is presented.

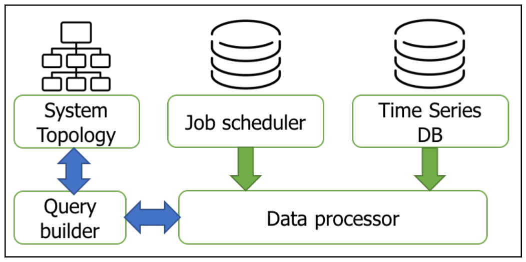


Figure 6.1: High-level data processing scheme

6.3 Predictability of an HPC Environment: Data Mining

6.3.1 General statistics

According to the data from the job scheduler, more than 1.3M jobs, and more than 270k different job names were submitted in the 10-months period represented by the dataset extracted from the Mistral production environment. These submissions, which are mainly executed in batch mode (98.8%), resulting in over 4.8M steps. Completed jobs are 91.3% of all submitted ones. In contrast, 5.6% of started jobs result in the fail state, 1.7% of submissions are cancelled, 1.4% result in timeout, and 0.028% fail because of computing node problems. Through the analysis of these data, it is observed that the mean number of allocated nodes is 3.4 for completed steps and 18 for failed ones. This result follows a typical pattern usually reported in state of the art: failed steps are usually more complicated. Average duration and standard deviation of failed jobs and completed ones are quite similar. When it comes to steps, completed ones take in average 414 s, while failed almost three times more. For detailed statistics, see Table 6.1 for jobs and Table 6.2 for steps. These general statistics represent a convincing motivation for generating savings with the early termination of jobs predicted to fail. An average failed job consumes many more CPU hours than completed one and decreases resources availability. About 1.2M of all steps from the set run for more than 60 s and 1.1M more than 120 s.

Table 6.1: Jobs statistics by Slurm state

State	count	Allocated nodes						Duration [s]					
		mean	SD	min	50%	75%	max	mean	SD	min	50%	75%	max
CANCELLED	23087	25	100	1	5	16	3264	2680	14952	1	310	1591	1.6M
COMPLETED	1238585	12	34	1	6	16	3276	1954	4190	1	419	1953	0.3M
FAILED	75897	15	39	1	6	16	1700	1763	4288	1	164	2979	0.4M
NODE_FAIL	390	67	289	1	10	46	3264	11087	105332	38	2472	6202	2.1M
TIMEOUT	17864	16	57	1	1	16	3078	11586	18140	60	2408	28803	0.6M
ALL	1355823	13	37	1	6	16	3276	2085	5444	1	425	2001	2.1M

Table 6.2: Steps statistics by Slurm state

State	count	**Allocated nodes**				**Duration** [s]				**Ave Disk Read** [GB]				**Ave Disk Write** [GB]			
		mean	SD	min	max	mean	SD	min	max	mean	SD	min	max	mean	SD	min	max
CANCELLED	53579	28	87	1	3264	3322	8679	1	183k	16	151	0	7821	3	37	0	2341
COMPLETED	4853842	3.4	17	1	3276	414	1902	1	235k	1	10	0	6993	0.2	5	0	30742
FAILED	197704	18	28	1	3249	1111	5273	1	346k	3	73	0	6629	0.2	15	0	4078
ALL	5105125	4.2	20	1	3276	471	2326	1	346k	1	23	0	7821	0.2	7	0	4078

6.3.2 Job state sequences

Outcomes from previous analysis encourage the analysis of correlations between user's past jobs and the final state of a subsequent job. Firstly, we create a matrix presenting job state transitions. In details, Figure 6.2 illustrates states of 2-jobs sequences, grouped by a user name and job name (exact string match). Another possibility to build these sequences is to match jobs by parts of their names, *e.g.*, without suffixes, which usually stand for a simulated year, or another parameter of a run application. Previous state NONE refers to initial submissions, from which 88% completes, and the majority of the rest fails. Importantly, only 19% of next submissions complete after a job failed and 75% of them still fail. Majority of jobs completes after a hardware failure of a node. Also, these data reveal important rationales. For instance, users often submit applications which are correct and do not fail. Then they start trails, implement changes, or merely develop their models. Majority of next submissions completes, but still, failures are two times more probable than cancellations or timeouts. A typical user is more likely to have a job in the completed state after it is cancelled than it is failed. An interesting fact is that the probability of a node failure reaches its maximum value after another node failure, and it has the same order of magnitude for all other states. Moreover, we present mean time between subsequent submissions in Figure 6.3 with corresponding standard deviation in Figure 6.4.

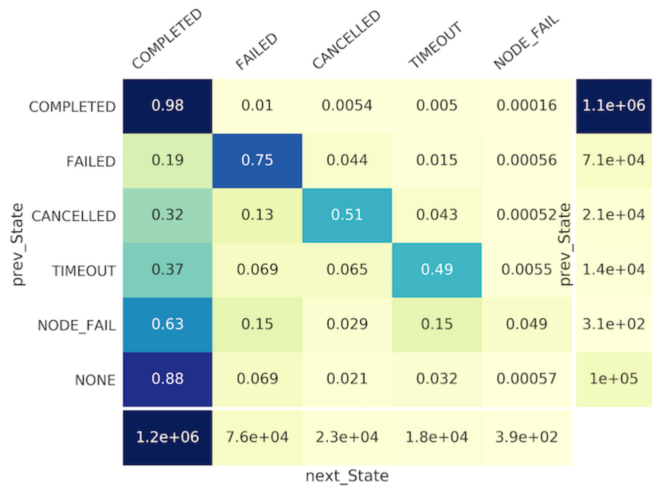


Figure 6.2: Heat-map presenting transition between 2 subsequent jobs, grouped by a user name and job name. For instance, after 0.32 of all jobs which are cancelled, the next jobs are completed ones

Regarding the correlation between cancelled and failed, 13% of next submissions after cancellations fails and only one third completes. Moreover, Table 6.2 shows that cancelled steps are characterized by much higher disk RW than completed and even failed ones. One of the potential causes after

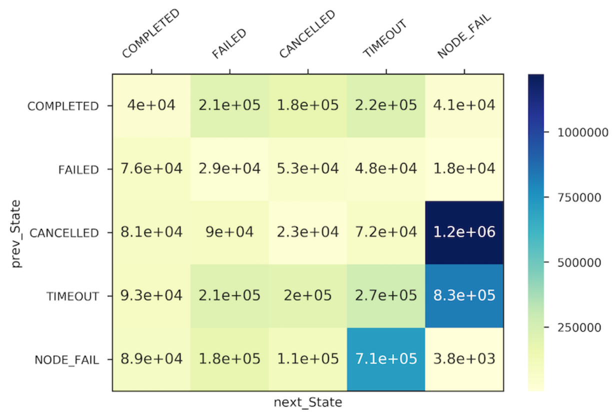


Figure 6.3: Heat-map presenting mean time [in seconds] between subsequent job states, grouped by user, application name.

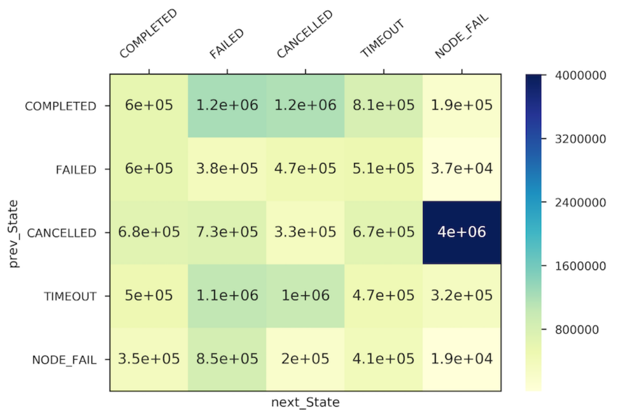


Figure 6.4: Heat-map presenting SD [in seconds] between subsequent job states, grouped by user, application name.

interviewing system administrators is that they cancel steps, due to high storage system usage – IO counters. Naturally, after cancellation, a job is possibly corrected and re-submitted to be completed. Further analysis is shown in Figure 6.5 which presents average factors of past failed and cancelled jobs to all submitted jobs in different N number of prior submissions for each job state. A readable observation is that, on average, in the preceding ten jobs there are as many cancelled jobs as failed ones for all states except node fail - probably lack of diverse samples. It can be highlighted that a cancellation often follows up other cancellations and a failure - other failures.

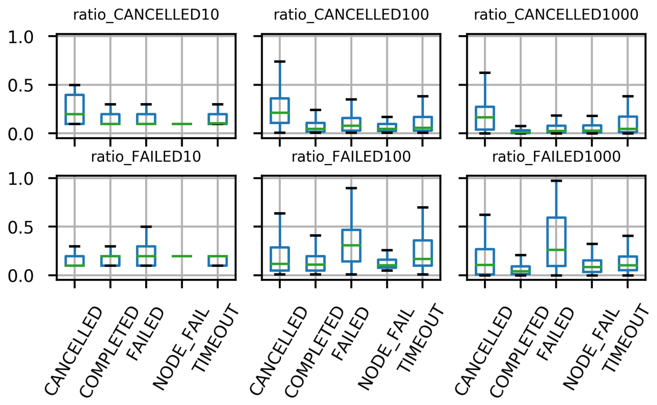


Figure 6.5: Plot presenting distributions of users' factors of failed and cancelled jobs for last N=10, 100, 1000 submissions for each succeeding job state. Users with more than 10 jobs submitted are counted. For instance, before failed job, a max. factor of failed in window of last N=10 submissions is 0.5.

Besides, in Figure 6.6, we present correlation type distribution between the number of failed and cancelled jobs in different time windows. Aggregation in 4-week periods and no lag (no shift between series) between these sequences reveals the highest number of sequences with correlation coefficient over a fixed threshold of 0.3. Additionally, we present distributions of a correlation coefficient value, see Figure 6.8 for different time windows. These distributions show that correlations are stronger for longer periods – weeks over days. In link with this, sequences of cancellations and failures are presented in Figure 6.7 for a randomly chosen user with relatively high activity. Surprisingly, it is observed that local minima of failed and cancelled jobs exist in the same time periods. In contrast, high activity of a user does not necessarily mean a high number of failures and cancellations. Naturally, a user might submit the same working code. These sequences reveal that there are periods of re-running the same models, and periods of experiments when a model is changed. This phenomenon is confirmed by researchers working in DKRZ.

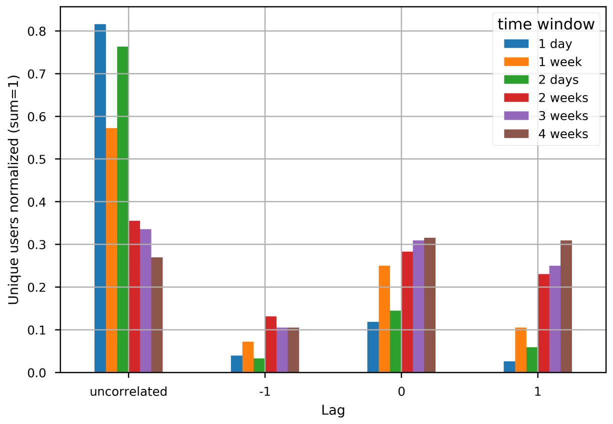


Figure 6.6: Plot presenting distribution of Pearson correlation coefficient for users with min. 1000 jobs submitted, correlation is counted for coefficients > 0.3. Total 304 users.

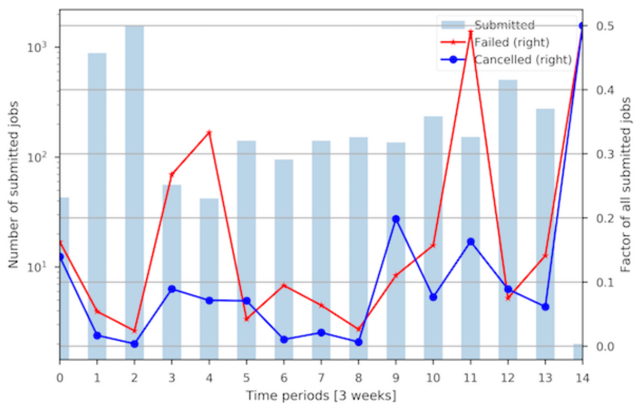


Figure 6.7: Plot presenting cancelled and failed job sequences aggregated in 3-week periods, for a relatively active user.

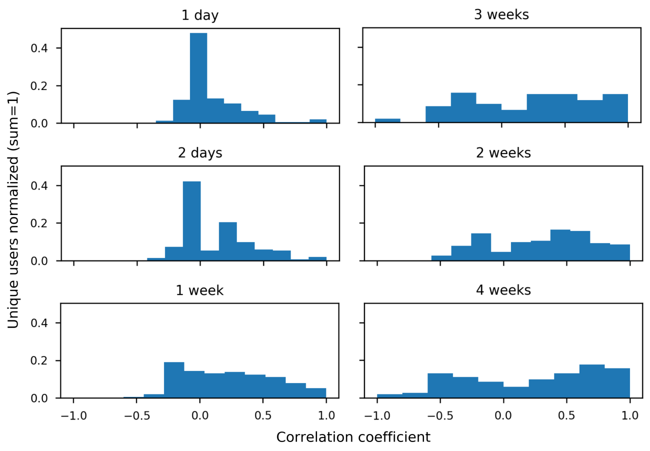


Figure 6.8: Plot presenting cancelled and failed jobs Pearson correlation coefficient distribution for users by aggregation periods

6.3.3 Time view

The overall cycle of jobs depending on the daytime can be seen in Figure 6.9. The number of jobs by the state is normalized to the mean number of started jobs during the whole daytime. Naturally, during the night the number of started jobs is much lower. Between 10 and 17 hour, the number of submissions is over the mean. Moreover, in Figure 6.11, we present distribution of time elapsed from job submission to a job start. This distribution shows that the highest waiting time is for jobs resulted in a timeout and node fail state.

In Figure 6.10, we present the average number of cancelled and failed jobs aggregated by daytime. It is clear that the highest number of failed ones starts between 14 and 16, while for cancelled the maximum is at 15 hour.

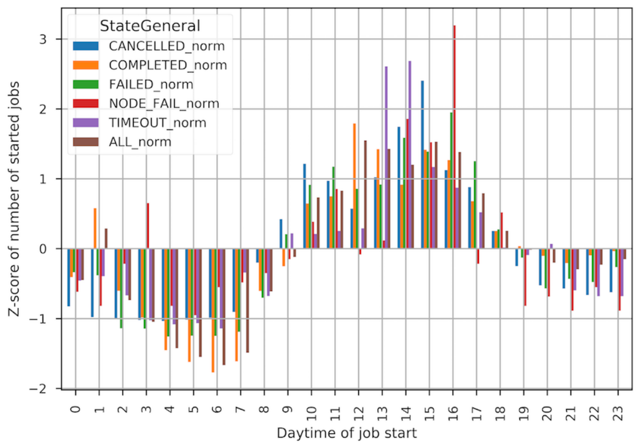


Figure 6.9: Plot presenting cancelled and failed jobs depending on the daytime of a job start

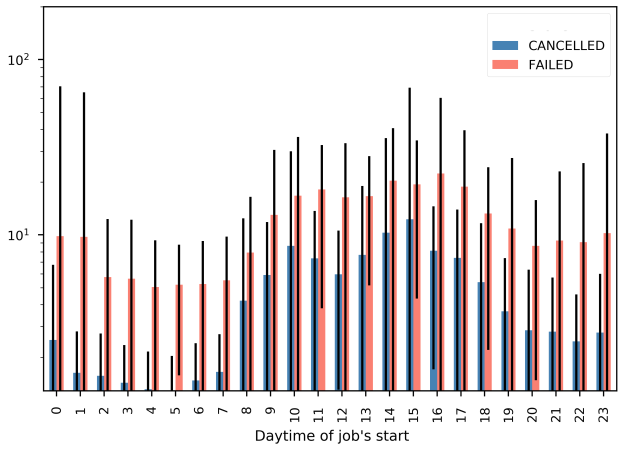


Figure 6.10: Plot presenting daily mean number (with stddev) of jobs finished as cancelled or failed by the daytime of a job start

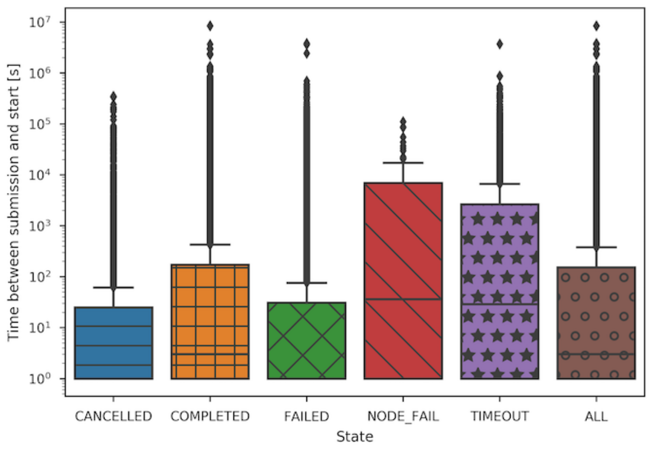


Figure 6.11: Plot presenting distributions of waiting time of submissions by state

6.3.4 Distribution of a job over the data center

Topology-aware resources allocation is applied as well as in Slurm, and other schedulers, see [109–111]. The job scheduler is optimized to use nodes which are closest to each other to reduce latency in data transfer. An interesting aspect to explore might be the distribution of the jobs over racks. Through this, we can discover the dependency between the number of network hops and failed jobs. The number of hops represents the complexity of a network topology for a particular job and increases with the number of used racks since a switch is mounted in each chassis. For this, we choose a subset of steps allocated on more than one node with duration more than 60 s. In average, completed steps are allocated on 1.1, $\sigma = 0.8$ racks, cancelled on 2.3, $\sigma = 2.8$ and failed on 1.8, $\sigma = 1.8$. Completed steps are not only distinguished by the lowest number of used racks, but also the lowest number of allocated nodes, as seen in Table 6.2.

The mean number of racks used by multi-node steps is 1.92. This distribution is presented in Figure 6.12. This figure also shows the probability of a failure according to the number of racks used for a step, and the maximum is at seven racks. For the number of racks over 13, which means using even more than 1000 computing nodes, occurrences of failures are rare. This phenomenon can be explained rather by a user's behavior than hardware dependencies. Most of HPC jobs are projected to be run on a specific number of nodes. This dependency is opposite to Big Data business software, where horizontal scaling on demand is one of the most important requirements in an application. So, the code for huge HPC jobs seems to be better tested and reliable for a fixed number of nodes.

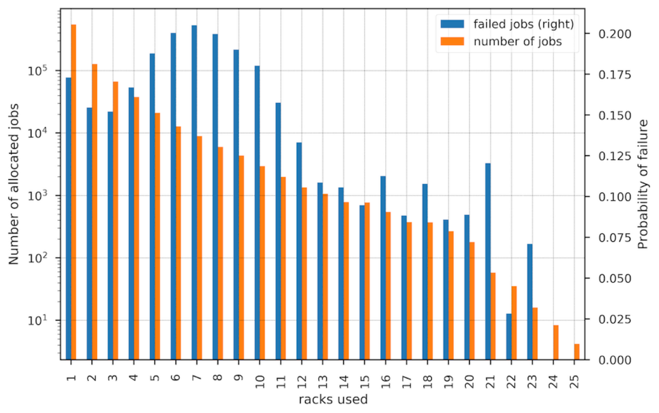


Figure 6.12: Plot presenting number of racks used for allocations for all steps and failed steps. N=841k

In Figure 6.13, we analyze duration over the number of racks used by a step. Notably, failed steps are statistically shorter than completed, when approximately less than ten racks are used for a step.

In this case, failures occur probably in the early phase of executed code. However, for the number of racks larger than 12, duration of failed steps significantly increases, while for completed ones it is kept on the same level. In Figure 6.14, distribution of the number of allocated nodes versus the number of racks can be seen. This relation is linear, although, in range of 10 and 20 racks used, the median number of allocated nodes does not increase. Completed steps with less than 100 nodes used are often placed in less than ten racks. It is opposite to failed or cancelled steps. The cancelled and failed ones are sparser, and for a few nodes allocated often use more racks.

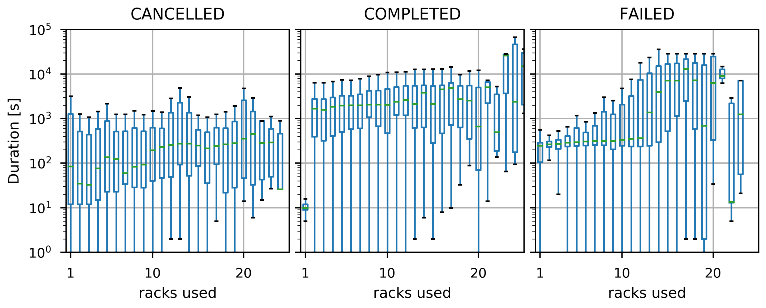


Figure 6.13: Plot presenting duration depending on number of racks used for a step

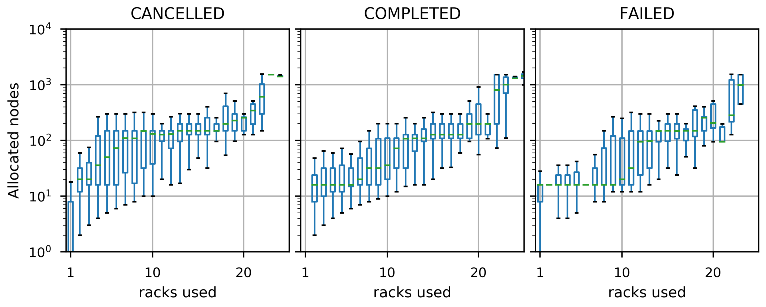


Figure 6.14: Plot presenting number of allocated nodes depending on number of racks used for a step

6.3.5 Node-power analysis

We investigate the power statistics of failed jobs in comparison with completed ones. Each computing blade is controlled and monitored by an isolated blade management controller which delivers **power metrics**. A controller is an external unit and acquiring measurements does not interfere with the workload of a blade. Power metrics of these blades perfectly depict their CPU load. Although in Subsection 6.4.1 we evaluate the usefulness of power statistics in predictions, we might also evaluate whether these series can improve job state prediction during the run-time.

We correlate power series of nodes allocated for a step with this the final state of this step and types of allocated nodes. In Table 6.3 we present average blade power and average last registered power for different job submissions states. In Table 6.4, we present power statistics for steps longer than 1000 s, grouped by hardware profile. The table shows average values of power metrics in the last 300 s. This value is lower for completed steps than for failed ones for all hardware groups. for some, it is even 15% difference. The most probable explanation can be the fact that once a software failure occurs some of the nodes go to an idle state.

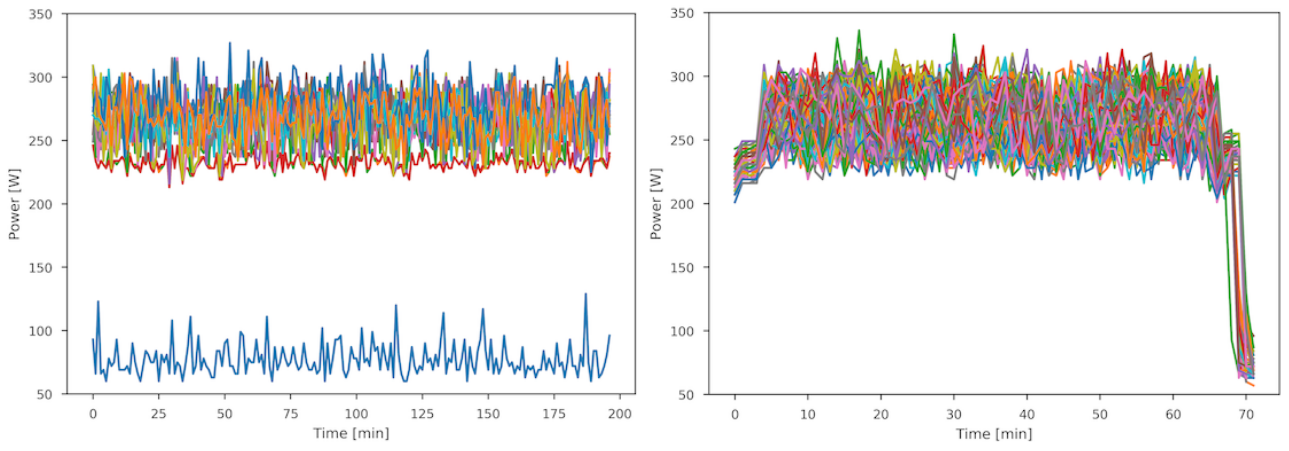
Table 6.3: Power statistics depending on a submitted job state, for submissions longer than 120 s

Job finish state	**Average blade power [W]**	**Average last registered blade power [W]**
Completed	265	228
Failed	242	227
Cancelled	240	203
Node failed	226	190

Table 6.4: Avg power in last 300 s of a job, partitioned by a job and node, for jobs longer than 1000 s, then aggregated

Profile	**State**	**avg(last power avg300) [W]**	**stddev(last power avg300) [W]**	**factor of COMPLETED**
B720-compute_36_64	CANCELLED	196	76	1.11
B720-compute_36_64	COMPLETED	176	82	1.00
B720-compute_36_64	FAILED	172	71	0.97
B720-compute_36_256	CANCELLED	209	75	1.00
B720-compute_36_256	COMPLETED	210	82	1.00
B720-compute_36_256	FAILED	186	76	0.89
B720-compute_36_128	CANCELLED	198	79	1.16
B720-compute_36_128	COMPLETED	170	82	1.00
B720-compute_36_128	FAILED	167	74	0.98
B720-compute_24_64	CANCELLED	225	92	0.95
B720-compute_24_64	COMPLETED	239	107	1.00
B720-compute_24_64	FAILED	190	113	0.80
B720-compute_24_256	CANCELLED	269	134	0.97
B720-compute_24_256	COMPLETED	277	154	1.00
B720-compute_24_256	FAILED	242	154	0.87
B720-compute_24_128	CANCELLED	248	116	0.93
B720-compute_24_128	COMPLETED	266	141	1.00
B720-compute_24_128	FAILED	227	142	0.85

For instance, Figure 6.15 presents power series of 1-step jobs, both executed with the same configuration by the same user. This scenario represents a typical case where one node is in an idle state, and the rest are executing some workload. On the contrary, power series of nodes executing a completed step do not show any node in an idle state. This phenomenon appears in other cases in the dataset and suggests that using power metrics would be relevant for classification of a job state. Moreover, this observation matches with the expert knowledge at DKRZ. In words of one of its system engineers: "We check the idle state of a node during a problematic job, looking at InfiniBand traffic of nodes. If it is low, a job is likely to fail."



(a) Failed **(b)** Completed

Figure 6.15: Plots presenting power series of 198 nodes running in parallel a job from the same, user, project, and application. Two jobs were run in different points of time. First one is failed, the next one is completed.

6.3.6 Additional analysis

During this research we analyzed other issues, which are not presented in previous sections, but are valuable to notice. Firstly, we evaluated heat exchange between blades, to check if there is any correlation between the temperature of blades placed in the same chassis. Probably because of high-performance cooling infrastructure, no relationship is discovered. Another considered issue is the priority of a job submission in relation to its final state. No apparent correlation is observed, although an anomaly is detected in the distribution of priority level for the timeout state. Comparing to other states, a normalized frequency of submissions with high priority is significantly higher for timeouts.

6.4 Mining important features and predicting the final job state with Decision Trees

According to the data from the job scheduler, more than 1.3M jobs, and more than 270k different job names are submitted in the 10-month period that is represented by the dataset extracted from the Mistral production environment. These submissions, which are mainly executed in batch mode (98.8%), result in over 4.8M steps. One of the observations from the statistics is coherent with the usual state of the art reports - failed steps are usually more complex [55]. These statistics represent a convincing motivation for generating savings with the early termination of jobs that are predicted to fail. An average failed job consumes many more CPU hours than completed one and it also decreases resources availability.

6.4.1 Most meaningful features for prediction of job states

Extraction of features. We generate Decision Trees (DTs) [112] to reveal job and step features explaining a job state. These ML models learn *if-then-else* rules, for either classification or regression task. An advantage of using a DT is the fact that it is a white-box model so that a human can easily understand a trained tree. We use all the features from each dataset for generation of a DT. To decide the optimal size of DTs, we consider (1) over-fitting and (2) readability of a model to a human. Firstly, we split our set into three sets using random stratified sampling. We create the training set containing 70% of jobs (samples), the validation set that has 10% of jobs (samples), and the test set with 20% of jobs. During the training, we measure accuracy on the validation set, while increasing depth of a tree. We set 100 as the minimum number of instances each node's child must have after a split. Trees with depth 5 obtain satisfactory performance. For larger DTs, the accuracy increase is low (0.03%), and the increase of the number of nodes is high. For instance, a tree with depth 9 has 275 nodes, and it is 84 nodes more than a DT with depth 8. Thus, we choose the optimal depth of the DT to be 5, which has 63 nodes. To check if models are not over-fitting, we evaluate random forests (RF) for each dataset. RF creates DTs and trains them with different training sets that are subsets of the main training set. Then, the results of each DT are combined. In our case, using RFs improve neither classification nor prediction quality when compared to the above DTs.

The test evaluations show the fitness of generated models of classification (having all information about a finished job), and prediction (having only information at the time of submission). We present the results in Table 6.5 and Table 6.6. We include only features with importance greater than 3%.

Table 6.5: Decision trees – evaluation of different combinations of data sets - **jobs**

Data set	**Important features**	**Job state**	**Completed**	**Failed**	**Cancelled**	**Timeout**	**Node fail**
configuration(C)	time limit (74%) daytime (24%)	precision	0.91	0.0	0.0	0.0	0.0
		recall	**1.0**	0.0	0.0	0.0	0.0
		f1-score	0.96	0.0	0.0	0.0	0.0
configuration+ user's history (C + U)	previous job state for a user (96%) number of allocated nodes (3%)	precision	**0.97**	0.75	0.52	0.68	0.0
		recall	0.98	0.70	**0.44**	0.30	0.0
		f1-score	0.98	0.72	**0.48**	0.42	0.0
statistics+ configuration+ user's history (S + C + U)	previous job state (87%) duration (9%) number of allocated nodes (4%)	precision	**0.97**	0.77	**0.63**	**0.81**	0.0
		recall	0.99	**0.74**	0.36	**0.35**	0.0
		f1-score	**0.98**	0.75	0.46	**0.49**	0.0
duration>120 s statistics, configuration, user's history (S + C + U)	previous job state (85%) duration (8%) number of allocated nodes (6%)	precision	**0.97**	**0.81**	0.62	0.80	0.0
		recall	0.99	**0.74**	0.31	0.33	0.0
		f1-score	**0.98**	**0.77**	0.41	0.47	0.0

Table 6.6: Decision trees – evaluation of different combinations of data sets - **steps**

Data set	**Important features**	**Job state**	**Completed**	**Failed**	**Cancelled**
configuration(C)	number of allocated nodes (98%)	precision	0.95	0.50	0
		recall	**0.99**	0.07	0
		f1-score	0.97	0.12	0
configuration, statistics (C + S)	number of allocated nodes (47%) average disk W (40%) duration (4%)	precision	**0.98**	0.83	0.58
		recall	**0.99**	0.76	0.04
		f1-score	**0.99**	0.79	0.79
duration>120 s configuration, statistics(C + S)	average disk W (47%) number of allocated nodes (36%) average CPU frequency (9%); duration (4%)	precision	0.95	0.59	**0.89**
		recall	0.98	0.23	**0.83**
		f1-score	0.97	0.33	**0.86**
configuration, topology, hardware information (C + T + H)	number of allocated nodes (79%) number of nodes 36C 64GB RAM (15%) number of nodes 36C 128GB RAM (3%)	precision	0.97	0.75	0.49
		recall	0.98	0.41	0.01
		f1-score	0.98	0.53	0.01
configuration, statistics, topology, hardware information, power statistics (C + S + T + H + P)	number of allocated nodes (46%) average disk W (41%); average disk R (5%) average VM size (4%)	precision	**0.98**	0.85	0.52
		recall	**0.99**	0.75	0.10
		f1-score	0.98	0.80	0.17
duration>120 s, (C + S + T + H + P)	average disk W (49%) number of allocated nodes (35%) average CPU frequency (10%)	precision	0.94	**0.93**	0.81
		recall	**0.99**	**0.79**	0.13
		f1-score	0.97	**0.85**	0.22

Jobs. The above results show that the size of the resource reservation is a principal factor determining the final state of a job. Also, the results expose that final states are highly correlated with a user's history. In general, this correlation is weaker for longer jobs.

Steps. Generated DTs reveal that the sum of *disk RW* is often higher for completed jobs than failed ones. Since the mean duration of failed steps is much higher than completed ones, see Table 6.2, higher storage usage can be explained by less active nodes in failed steps. We can state a hypothesis, that some nodes in failed steps stay in idle state, see Section 6.3.5. The evaluation shows the high importance of a number of allocated nodes with 36 cores. An upgrade done in DKRZ explains this phenomenon. The dataset includes the period when Broadwell nodes started their service in the production environment. That time, users were translating their software and scripts to the recently installed hardware. It is the primary cause of many job failures.

Conclusions. The evaluation of DT classification tasks reveals that a DT model is unable to classify and predict cancelled, node failed (0% for all of the data sources), or timeout jobs based only on configuration data. These data are the only information known to the scheduler after a job is submitted. The *f1-score* is 0 for all of the mentioned states. Augmenting this set with past user's submissions improves recall of failed jobs to 72% and lifts the precision of predicting cancellations to 52% and timeouts to 68%. This result shows a strong correlation inside a sequence of final job states. Adding to the training dataset metrics which are known after a job is finished increases the precision of a classifier. The recall does not change for any of the states. Regarding steps, precision and recall are lower than those for job submissions. It is a reasonable result considering that steps have a lower number of features available for these evaluations. The number of allocated nodes is an important feature to predict the final state of a job even when used with hardware metrics features. Other important features are knowledge on past submissions and their states. According to the hardware statistics, average *disk W* is a highly important feature in the classification task of final job states, while general power statistics are features with low importance. Note that according to steps, none of the data sources maximizes the prediction abilities of all of the states.

6.5 Prevention of failures through static and dynamic policy

Prior data exploration and evaluation of DTs show that power metrics and DTs can be used for prediction of final job state. Predictions contain probabilities for each step state. During prediction, we classify a step as failed, when the probability of failure is higher than a defined threshold and all other probabilities associated with other classes. Therefore, we propose two types of policies to be used: a **static** and **dynamic** one. A static policy uses predictions based on a step configuration data, topology, and hardware information ($C + T + H$) through DTs. A dynamic policy uses predictions during run-time which are produced by a convolutional neural network (CNN), introduced in Section 6.5.1. The inputs to this model are power metrics, which are analyzed in Section 6.3.5. While using a dynamic policy, a job is killed when it is classified as failed for the first time – the earliest prediction over the given threshold.

The use of different types of models, one as a white-box and the other as a black-box has several advantages over, for instance, one complex NN model trained with both static and dynamic data. Firstly, the use of DTs enables to easily explain phenomena observed in a data center to system administrators. Since a model can evolve by repeating the training, changes in trends and user behavior occurred in a data center are observed as results of the comparison of models. Also, a failure prevention system gains performance during the run-time because of splitting evaluation to offline (time of submission only) and online (evaluation of a job during its runtime) one.

6.5.1 Dynamic Policy: Preventing failures during runtime through Convolutional Neural Networks

CNNs are a type of deep neural network [113] following a design of biological vision systems [114]. They are widely used for image classification, natural language processing, and recommendation systems, and they have also been successfully used for time series classification and prediction. We propose to use a CNN for classification and prediction of multivariate time series, which are the power metrics of nodes (overall energy consumption of a computing blade) used in a step. Therefore, CNN learns "how a multivariate time series of nodes executing a step look like." A major advantage of using CNNs over neural nets with fully connected (dense) layers only, is that they need much fewer neurons and parameters to solve a particular classification or prediction problem. CNNs are able to learn features and share neuron weights, which minimizes their space complexity.

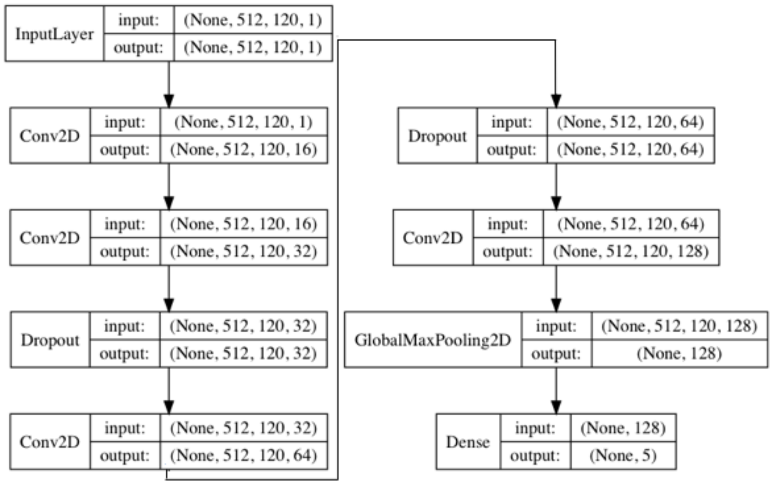

InputLayer
input: (None, 512, 120, 1)
output: (None, 512, 120, 1)
Conv2D
input: (None, 512, 120, 1)
output: (None, 512, 120, 16)
Conv2D
input: (None, 512, 120, 16)
output: (None, 512, 120, 32)
Dropout
input: (None, 512, 120, 32)
output: (None, 512, 120, 32)
Conv2D
input: (None, 512, 120, 32)
output: (None, 512, 120, 64)
Dropout
input: (None, 512, 120, 64)
output: (None, 512, 120, 64)
Conv2D
input: (None, 512, 120, 64)
output: (None, 512, 120, 128)
GlobalMaxPooling2D
input: (None, 512, 120, 128)
output: (None, 128)
Dense
input: (None, 128)
output: (None, 5)

Figure 6.16: Graph presenting a trained CNN with layers type and shape of the data

In Figure 6.16, we present the best CNN model trained for this task. We create the final model after a few iterations, through dropping layers from more complex models which over-fit during the training and do not increase the accuracy. The model presented in Figure 6.16 comprises a few types of layers. Each convolutional layer comprises filters with size 3x3, and during the training, each filter learns weights. This layer is used to extract specific features, in this case from 2D matrices. Another important layer type used is a drop-out, which regularizes weights and through dropping neurons and connections, prevent overfitting [115]. A max pooling layer and dense layer are used to aggregate extracted features and classify them into defined classes and give probabilities. The input data are 2D matrices of size M=512 (number of nodes) x T=120 (length of time series). For steps with matrices which shape is less than MxT, we pad a sample with zeros - which are ignored by CNN during the training. For these matrices which are larger than that size, we downsample a matrix by averaging power metrics. The value for T is chosen so that it is large enough to represent the complete series of most of the steps (only 1.3% of steps are longer than 120 min) and at the same time it is small enough for the NN training to be practical. The dataset with steps is split randomly (the same split as in Section 6.4) into three sets: training (70% of the data), validation (10%), and test (20%) respectively.

The CNN is trained using tensorflow[2] and keras[3] libraries by means of 2x GPU GeForce 1080 Ti. Also, after a few trails and examining a shape of the loss curve, the learning rate is set to 0.001, and we choose a stochastic gradient descent optimizer. The final model, which contains 32261 parameters to train, is trained in 67 epochs with approximately 1h per epoch. We stop training after lack of

[2]https://www.tensorflow.org/
[3]https://keras.io/

significant improvement in the loss curve, and when the model does not improve more than 1% in 5 epochs. We show results of the trained CNN in Table 6.7. We can see, that all classes are classified with the satisfactory level of precision: from 79% for cancelled to 93% for completed. However, the *f1-score* is acceptable only for failed (74%) and completed (96%) jobs.

6.5.2 Evaluation: Static and dynamic job-killing policies

The primary goal of the evaluation is to explore possible savings and losses depending on the aggressiveness of job-killing policy. We measure the aggressiveness of a policy as the *threshold of class prediction probability.* For instance, a threshold of 60% means that a job is classified as failed when the probability of predicting failed is higher than 60%. An aggressive policy is the one with a low threshold, and the less aggressive one is the one with a high threshold, *e.g.*, greater than 90%. We evaluate the trained CNN model and DT to predict the final states of steps. We use a test set which contains jobs with total CPU time of 84.7M h. CNN predicts a final job state and outputs probabilities for each timestamp during the run of a job. We evaluate proposed policies by depicting lost and gained CPU time, expressed in hours. Lost CPU time stands for the resources consumed by a step that is labeled as completed, but it is killed (false positive). Saved CPU time represents resources that would be used until a step ends but are saved due to a decision of early step termination. Approximate performance of CNN evaluation is 5000 samples/s which is considered sufficient for these experiments.

Table 6.7: CNN test results - Classification. Data set: steps – power metrics, duration > 120 s

	Completed	**Failed**	**Cancelled**
precision	0.93	0.85	0.79
recall	0.98	0.66	0.15
f1-score	0.96	0.74	0.25
test set	168875	28605	4457

Table 6.8: Summary of the dynamic policy evaluation over a test set containing 11M CPU hours of failed jobs

Dynamic policy metric	**CPU h**	**Probability threshold**
maximum savings achievable	7.9M	<0;0.42>
maximum loss (false positive)	4.1M	0.52
global maximum (savings - loss)	4.0M	<0;0.42>
local maximum of (savings - loss)	0.7M	0.82

Considering the dynamic policy, the maximum value of true positives is 0.9, and for false positives, the maximum value is 0.45. Both metrics decrease smoothly when the threshold grows. Figure 6.17 shows true and false positive rates depending on the probability threshold for failure prediction with the CNN. On the other hand, the static policy is characterized by the maximum value of the true positive rate of 0.47 and a small value of 0.02 for the false positive rate. The static policy is more accurate in predictions comparing to the dynamic one, but the maximum number of predicted steps to fail are almost two times lower.

Wastes. When it comes to the CPU time, the static policy allows for maximum savings of 0.8M CPU h, and the dynamic one (for threshold of the global maximum) of 8M CPU h. In Table 6.8, we present a summary of the evaluation of the dynamic policy considering CPU hours of jobs. Note, that the earlier we kill a failed job, the bigger savings are. On the other hand, the confidence of prediction increases with time a job is running as we gather more data. Regarding these trade-offs, there is a global maximum of losses for threshold 0.52. For instance, applying a dynamic policy with a threshold of 0.82 (local maximum with the highest threshold value) to the test dataset saves 1.6M CPU h with 0.9M CPU h lost and the total profit of 0.7M CPU h. For instance, a less aggressive policy would be the application of a threshold equal to 0.96. In this case, we save 210k CPU h, and we lose 24k CPU h, with the total profit of 190k CPU h. In contrast, executing static policy allows for maximum savings of 870k CPU h by killing 13k failed jobs with a side effect of killing 3.8k completed ones. Also, the application of the static policy, which is more conservative, does not cause a loss in CPU time, because it reacts after job submission.

Figure 6.18 presents the distribution of job time at which the dynamic policy will react and terminate a job. We can see that almost half of the jobs are killed during the first 30% of their total execution time (the time they take if they are not killed earlier). Then, for the remaining steps, prediction abilities increase after 60% of their duration. Figure 6.18 shows that the dynamic policy can predict failures early.

Users and system administrators may use policies with different aggressiveness levels. For instance, a user might choose a very aggressive policy, both static and dynamic with a very low threshold, when the project budget is highly limited. On the other hand, a less aggressive policy, *e.g.*, a dynamic policy with a high threshold, above 0.9, can be appropriate for long jobs, where user time is the most expensive factor to consider. Also, such a policy can maximize savings comparing to use of a static policy. A static policy used by system administrators can help eliminating problematic jobs, which may be causing the overload of a system. However, use of dynamic policy can cause dissatisfaction of users, since this policy can unexpectedly terminate their jobs without a known reason, even when

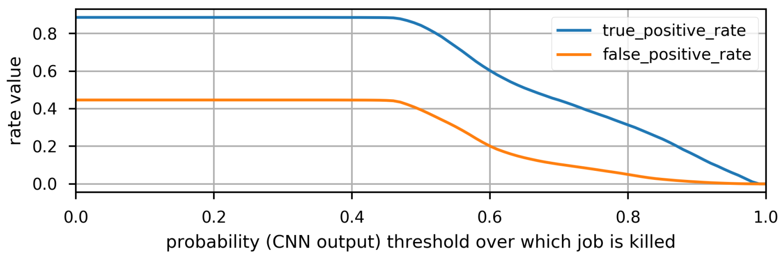


Figure 6.17: Plot presents the evaluation of CNN model for different values of prediction probability threshold. The lower is the threshold, the more aggressive is the job terminating policy, greater savings, but we kill more good jobs as a consequence of inaccurate predictions. Total CPU Hours of failed jobs in a set: 11M.

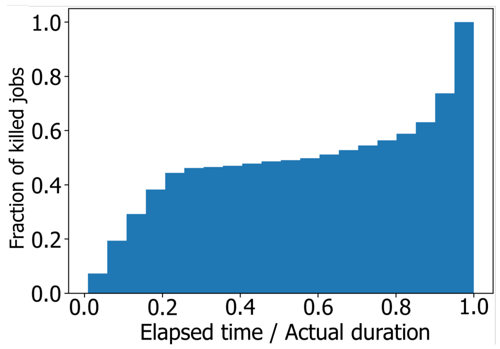


Figure 6.18: Cumulative plot presenting the time when the probability of failure exceeds defined threshold 0.82. N=7300

a user is not currently working with a system. If it happens with the static policy, a user gets a job terminated immediately after submission, when he is online.

Also, supervised learning through interaction with a user can help improving the proposed policies. Firstly, users should receive a notification when their jobs are repeatedly killed after re-submissions. A user or a system administrator could label such a problematic job. This action provides a model with additional information for incremental improvement. Also, system administrators can decide to perform supervised learning, to set up the optimal aggressiveness of the policy (threshold).

6.6 Discussion, conclusion and future work

In this chapter, we analyzed a dataset containing metrics, topology and job scheduler data for the Mistral supercomputer. We showed important features in a classification and prediction task of a job state. The number of allocated nodes, the state of a previous job submitted by a user, average storage writes are the most important ones. DTs detect specific node types as an important feature due to migration process from the old to the new computing nodes. DTs perform well as a classifier, with a recall nearly 80% and a precision of 93% for failed steps. As a predictor, DTs can point failed steps, using configuration and allocated hardware data exclusively, with a recall of 41% and a precision of 75%. In the case of CNNs, these scores increase to 66% and 85% respectively. This chapter shows that one of the biggest influences on the next state of a job in a supercomputer like Mistral lies in the diversity and spatial distribution of allocated nodes, place of a job in a user sequence and number of disk operations.

We evaluated dynamic and static job-killing policies, pointing out possible savings related to the aggressiveness of both policies. For instance, using medium-aggressive approach, we can kill more than 28% of failed jobs. Through CNN predictions, the proposed dynamic policy kills 40% of jobs in the first 20% of their duration. These effects can be improved by utilizing feedback from users and system administrators and adjusting weights of CNN by supervised learning.

As future work, we would like to improve prediction capabilities of the created solution. Firstly, we can achieve more accurate analysis of final job states by adding OS logs to the analyzed dataset. Also, this would help to build prediction algorithm of final job states, which is not limited by Slurm job state but uses the utility of a job. For instance, the utility can be measured by analyzing users' actions after a job finishes, *e.g.*, a user copied output data, re-run the same code with different parameters, changed the code. Therefore, this approach can differentiate jobs with a non-zero return code from these which were run unnecessary and these which can provide any utility to a user, *e.g.*, development progress, part of results. Then, we can consider a more complex model which considers step sequence for a job. Also, we would like to consider additional input information such as real-time metrics from the data center, *e.g.*, Lustre I/O, overall system load and IB traffic. Finally, we would like to focus more on the deep learning algorithms for prediction of failures and root cause analysis.

CHAPTER 7

CONCLUSIONS

In this Thesis, we presented contributions related to diagnostics, root cause analysis, failure prediction, and prevention. We performed the research on different computing environments which are crucial for today and tomorrow IT systems. Specifically, we explored different research problems such as

- scalability of RCA
- dynamism of a diagnosed environment
- integration of logs and metrics to represent system state in the form of a graph
- predictability of a supercomputing environment
- prevention of failures to save resources and user time

We used diverse computing environments for our research: the petascale Mistral supercomputer, simulated IoT environment, on-premise Big Data cluster, and containers.

Thanks to the research we:

- Proposed a new fast root cause analysis framework which works on an optimized transformation of Bayesian networks to Arithmetic Circuits. The system performs orders of magnitude faster while using fewer memory resources while comparing to conventional approaches. The system can perform approximate inference in Bayesian networks with the size of millions of nodes.
- Proposed an Actor-Based root cause analysis System which can work well in highly distributed and dynamic environments. The system performs root cause analysis using self-diagnostics concept, which helps to operate in, *e.g.*, environments with connectivity problems.
- Proposed a weighted graph-based framework for root cause classification and diagnostics through knowledge transfer. The framework represents a system state in the form of a graph, including

logs, metrics and other information, *e.g.*, hardware or software component types. The framework finds the closest neighbor graph through the evaluation of graph similarity. The framework is proven to perform root cause classification through knowledge transfer, achieving *f1-score* of 0.77.

- Explored trends, and critical job features deciding on an HPC job final state. Then, we proposed failure prevention policies: dynamic and static one. The static one works on Decision Trees and different data sets containing historical information on jobs and the infrastructure. It allows evaluating the likelihood of failure at the time of submission. The dynamic policy works on historical power series of multinode jobs and Convolutional Neural Networks.

Further development of contributions presented in this thesis can be done in several dimensions. First of all, actor-based root cause analysis can be deployed on a real Internet of Things environment. There are existing small and medium-size test-bed platforms containing tens up to hundreds of elements. However, the challenge is a deployment of ABRCA on millions of intelligent devices. Then, the experiments can include exploration of the impact of the algorithm to the standard functionality of IoT devices. Future work on the algorithm can be focused on the resource decision system. Such a system should take optimal decisions on delegating the stages of the calculations to appropriate devices and possibly subnetworks of the main network of devices. The resource decision system should consider different constraints and consequences of utilizing various resources, *e.g.*, network, CPU, battery. Example constraints are latency, energy consumption, and performance degradation.

Next crucial future work is the application of knowledge transfer framework on large scale real environment. There might be interesting research performed on knowledge transfer framework integrated with knowledge exploration solutions. Such a system could automatically mine knowledge on failures from parts of the system. The solution might be combined with root cause analysis methods proposed in this thesis. Another vital issue to consider in the future work is an automatic taxonomy construction. It can be elaborated starting from host discovery techniques and using more complex techniques such as mining application and host logs and tracing application calls. Then the knowledge transfer would be much more automated. Also, the work can be focused on the proactive maintenance.

The further steps according to work on failure prevention policies it can be transferring models trained on the Mistral dataset to other supercomputing facilities. Interesting research can be performed to create a universal and transferable model for failure prediction. Then a model could be additionally trained on specific local data sets.

We believe that thanks to our contributions we do not only increase the performance of diagnostics and provide automated tools. We think that thanks to our work the future solutions will be able to limit the amount of work for the human. We should now focus on aiding the management of systems which are so complex that a human is already not able to maintain them manually.

DISSEMINATION ACTIVITIES

Publications

1. **Michał Zasadziński**, Marc Solé, Alvaro Brandon, Victor Muntés-Mulero and David Carrera. Next Stop "NoOps": Enabling Cross-System Diagnostics Through Graph-based Composition of Logs and Metrics. *IEEE Cluster 2018.* [**A ranked conference**]

2. **Michał Zasadziński**, Victor Muntés-Mulero, Marc Solé, David Carrera and Thomas Ludwig. Early Termination of Failed HPC Jobs Through Machine and Deep Learning. *Euro-Par, 24th International European Conference on Parallel and Distributed Computing.* 2018 [**A ranked conference**]

3. **Michał Zasadziński**, Victor Muntés-Mulero, Marc Solé and Thomas Ludwig. Mistral Supercomputer Job History Analysis. 2018. *https://arxiv.org/abs/1801.07624.* [Technical Report]

4. **Michał Zasadziński**, Victor Muntés-Mulero, and Marc Solé Simo. Actor based Root Cause Analysis in a distributed environment. *In Proceedings of the 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS '17).* IEEE Press, Buenos Aires (Argentina), May 21, 2017. Pages 14-17

5. **Michał Zasadziński**, Victor Muntés-Mulero, Marc Solé and David Carrera. Fast Root Cause Analysis on Distributed Systems by Composing Precompiled Bayesian Networks. *In Procs. of the International Conference on Machine Learning and Data Analysis 2016 (ICMLDA).* Berkeley. October 2016 [**Best student paper award**]

6. Athanasios Kiatipis, Alvaro Brandon, Rizkallah Touma, Pierre Matri, **Michał Zasadziński**, Linh Nguyen, Adrien Lèbre and Alexandru Costan. A Survey of Benchmarks to Evaluate Data Analytics for Smart-* Applications. *IEEE Transactions on Big Data. Special Issue on Edge Analytics in the Internet of Things.* 2018 [unpublished, under review]

Patents

1. **Michał Zasadziński**, Victor Muntés-Mulero, Marc Solé Simo. Protection system against illegitimate resource usage by websites to exploit peer to peer software. US Patent Application 16/046143. filed to USPTO July 26th, 2018.

2. **Michał Zasadziński**, Marc Solé Simo, Victor Muntés-Mulero. Fault tolerant Root Cause Analysis system. US Patent Application 15/485848. filed to USPTO April 12th, 2017

3. **Michał Zasadziński**, Marc Solé Simo, Victor Muntés-Mulero. Model Based Root Cause Analysis. US Patent 20170372212A1 15/195916. filed to USPTO June 28th, 2016

LIST OF REFERENCES

[1] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[2] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[3] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[4] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. *NetDB*, 2011.

[5] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[6] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[7] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward black-box online fault localization for cloud systems. In *IEEE 33rd International Conference on Distributed Computing Systems*, pages 21–30, July 2013.

[8] C. Wang, K. Schwan, B. Laub, M. Kesavan, and A. Gavrilovska. Exploring graph analytics for cloud troubleshooting. In *11th International Conference on Autonomic Computing*, 2014.

[9] G. Bronevetsky, I. Laguna, B. R. de Supinski, and S. Bagchi. Automatic fault characterization via abnormality-enhanced classification. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012.

[10] M. Miyazawa and K. Nishimura. Scalable root cause analysis assisted by classified alarm information model based algorithm. In *2011 7th International Conference on Network and Service Management*, pages 1–4, Oct 2011.

[11] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 27–27, Berkeley, CA, USA, 2011. USENIX Association.

[12] S. A. Yemini, S. Kliger, and E. Mozes. High speed and robust event correlation. *IEEE Communications Magazine, pp*, pages 82–90, 1996.

[13] H. Madsen, B. Burtschy, G. Albeanu, and F. Popentiu-Vladicescu. Reliability in the utility computing era: Towards reliable fog computing. In *20th International Conference on Systems*, Bucharest, 2013. Signals and Image Processing.

[14] C. C. Aggarwal. Managing and mining sensor data. *Springer pp*, 396:9–46, 2013.

[15] A. Vahid Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya. *Fog Computing: Principles, Architectures, and Applications.* Principles and Paradigms, Massachusetts, in Internet of Things, 2016.

[16] O. Niggemann, G. Biswas, J. S. Kinnebrew, H. Khorasgani, S. Volgmann, and A. Bunte. *Data-Driven Monitoring of Cyber-Physical Systems Leveraging on Big Data and the Internet-of-Things for Diagnosis and Control.* in Proceedings of the 26th International Workshop on Principles of Diagnosis, Lemgo, 2015.

[17] Teodora Sanislav and Liviu Miclea. Cyber-physical systems-concept, challenges and research areas. *Journal of Control Engineering and Applied Informatics*, 14(2):28–33, 2012.

[18] J. Shi, J. Wan, H. Yan, and H. Suo. A survey of cyber-physical systems. In *Int. Conf. on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, Nov 2011.

[19] K. Liu, Q. Ma, X. Zhao, and Y. Liu. Self-diagnosis for large scale wireless sensor networks. In *2011 Proceedings IEEE INFOCOM*, pages 1539–1547, April 2011.

[20] Oliver Niggemann, Stefan Windmann, Sören Volgmann, Andreas Bunte, and Benno Stein. Using learned models for the root cause analysis of cyber-physical production systems. In *Int. Workshop Principles of Diagnosis (DX).* Graz, Austria, Sep 2014.

[21] Y. Zhang, I. L. Yen, F. B. Bastani, A. T. Tai, and S. Chau. Optimal adaptive system health monitoring and diagnosis for resource constrained cyber-physical systems. In *20th Int. Symp. on Software Reliability Engineering*, pages 51–60, Nov 2009.

[22] S. P. Kavulya, K. Joshi, F. Di Giandomenico, and P. Narasimhan. *Failure Diagnosis of Complex Systems.* Springer Publishing Company, Incorporated, 2012.

[23] J. Shumann, T. Mbaya, O. Mengshoel, K. Pipatsrisawat, A. Srivastava, A. Choi, and A. Darwiche. Software health management with Bayesian networks. *Innovations in Systems and Software Engineering*, 9(4):271–292, December 2013.

[24] E. Kiciman, D. Maltz, and J. C. Platt. *Fast Variational Inference for Large-scale Internet Diagnosis.* in Advances in Neural Information Processing Systems 20, 2007.

[25] L. Bennacer, Y. Amirat, A. Chibani, A. Mellouk, and L. Ciavaglia. Self-diagnosis technique for virtual private networks combining Bayesian networks and case-based reasoning. *IEEE Transactions on Automation Science and Engineering*, 12(1):354–366, January 2015.

[26] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.

[27] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning).* The MIT Press, 2007.

[28] Lionel Torti, Christophe Gonzales, and Pierre-Henri Wuillemin. Patterns discovery for efficient structured probabilistic inference. In Salem Benferhat and John Grant, editors, *Scalable Uncertainty Management*, pages 247–260, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[29] Y. Xiang, K. Olesen, and F. Jensen. *Some Practical Issues in Modeling Diagnostic Systems with Multiply Sectioned Bayesian Networks.* in Proceedings of the Twelfth International FLAIRS Conference, Florida, 1999.

[30] F. J. Díez and M. J. Druzdziel. *Canonical probabilistic models for knowledge engineering.* Madrid, Spain, 2006.

[31] A. Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50(3):280–305, May 2003.

[32] M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational Bayesian networks for exact inference. *Int. J. Approx*, 42:1–2, 2006.

[33] Ole J. Mengshoel, Adnan Darwiche, Keith Cascio, Mark Chavira, Scott Poll, and Serdar Uckun. Diagnosing faults in electrical power systems of spacecraft and aircraft. In *Proceedings of the 20th National Conference on Innovative Applications of Artificial Intelligence - Volume 3*, IAAI'08, pages 1699–1705. AAAI Press, 2008.

[34] S. Zermani, C. Dezan, R. Euler, and J. P. Diguet. Online inference for adaptive diagnosis via arithmetic circuit compilation of bayesian networks. *Designing with Uncertainty: Opportunities & Challenges workshop*, 2014.

[35] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP, pages 378–393, New York, NY, USA, 2015. ACM.

[36] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, May 2015.

[37] A. Abu-Samah, M.K. Shahzad, E. Zamai, and A.Ben Said. Failure prediction methodology for improved proactive maintenance using bayesian approach. *IFAC-PapersOnLine*, 48(21):844 – 851, 2015. 9th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS.

[38] Carmelo Cascone, Davide Sanvito, Luca Pollini, Antonio Capone, and Brunilde Sans. Fast failure detection and recovery in sdn with stateful data plane. *International Journal of Network Management*, 27(2):e1957, 2016.

[39] Daniel Hsu. Anomaly detection on graph time series. *CoRR*, abs/1708.02975, 2017.

[40] Caleb C Noble and Diane J Cook. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636. ACM, 2003.

[41] Timo Schindler. Anomaly detection in log data using graph databases and machine learning to defend advanced persistent threats. *CoRR*, abs/1802.00259, 2018.

[42] Laura A Zager and George C Verghese. Graph similarity scoring and matching. *Applied mathematics letters*, 21(1):86–94, 2008.

[43] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. Algorithms for graph similarity and subgraph matching, 2011.

[44] Panagiotis Papadimitriou, Ali Dasdan, and Hector Garcia-Molina. Web graph similarity for anomaly detection. *Journal of Internet Services and Applications*, 1(1):19–30, 2010.

[45] Wael H Gomaa and Aly A Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13), 2013.

[46] Aminul Islam and Diana Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2(2):10, 2008.

[47] Jan Wira Gotama Putra and Takenobu Tokunaga. Evaluating text coherence based on semantic similarity graph. In *Proceedings of TextGraphs-11: the Workshop on Graph-based Methods for Natural Language Processing*, pages 76–85, 2017.

[48] Ping Li, CX Wu, SZ Zhang, XW Yu, and HD Zhong. Mining users' preference similarities in e-commerce systems based on webpage navigation logs. *International Journal of Computers, Communications & Control*, 12(5), 2017.

[49] Sergio Flesca, Sergio Greco, Andrea Tagarelli, and Ester Zumpano. Mining user preferences, page content and usage to personalize website navigation. *World Wide Web*, 8(3):317–345, 2005.

[50] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Phil Cudre-Mauroux. Dependency-driven analytics: a compass for uncharted data oceans. Technical report, Microsoft, October 2016.

[51] Jie Lu, Vahid Behbood, Peng Hao, Hua Zuo, Shan Xue, and Guangquan Zhang. Transfer learning using computational intelligence: A survey. *Knowledge-Based Systems*, 80:14 – 23, 2015.

[52] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[53] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proc. of the 1st ACM Symp. on Cloud Comput.*, pages 193–204. ACM, 2010.

[54] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lect. on comput. architecture*, 8(3):1–154, 2013.

[55] Yulai Yuan, Yongwei Wu, Qiuping Wang, Guangwen Yang, and Weimin Zheng. Job failures in high performance computing systems: A large-scale empirical study. *Comput. Math. Appl.*, 63(2):365–377, 2012.

[56] N. El-Sayed and B. Schroeder. Reading between the lines of failure logs: Understanding how hpc systems fail. In *43rd Annu. IEEE/IFIP Int. Conf. on Dependable Syst. and Networks (DSN)*, pages 1–12, 2013.

[57] A. D. Clark, L. M. Tellez, S. Besse, and J. M. Absher. Dynamic prediction estimation of intentional failures in hpcs. In *Int. Conf. on Advances in Social Networks Anal. and Mining*, pages 1244–1250, 2016.

[58] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto. Online failure prediction in cloud datacenters by real-time message pattern learning. In *4th IEEE Int. Conf. on Cloud Comput. Technol. and Sci. Proc.*, pages 504–511, 2012.

[59] Matthew D. Jones, Joseph P. White, Martins Innus, Robert L. DeLeon, Nikolay Simakov, Jeffrey T. Palmer, Steven M. Gallo, Thomas R. Furlani, Michael T. Showerman, Robert Brunner, Andry Kot, Gregory H. Bauer, Brett Bode, Jeremy Enos, and William T. Kramer. Workload analysis of blue waters. *CoRR*, abs/1703.00924, 2017.

[60] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. *SC17*, pages 1–12, 2017.

[61] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surv.*, 42(3):10:1–10:42, 2010.

[62] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. Diagnosing performance variations in hpc applications using machine learning. In *High Performance Computing*, pages 355–373, Cham, 2017. Springer International Publishing.

[63] Bing Xie, Yezhou Huang, Jeffrey S. Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. Predicting output performance of a petascale supercomputer. In *Proc. of the 26th Int. Symp. on High-Performance Parallel and Distributed Computing*, pages 181–192, 2017.

[64] Jim Gao. Machine learning applications for data center optimization. *Google White Paper*, 2014.

[65] Song Fu and Cheng-Zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Proc. of Conf. on Supercomp.*, pages 41:1–41:12, 2007.

[66] J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, and B. H. Park. Dynamic meta-learning for failure prediction in large-scale systems: A case study. In *37th Int. Conf. on Parallel Process.*, 2008.

[67] Alina Sîrbu and Ozalp Babaoglu. Towards operator-less data centers through data-driven, predictive, proactive autonomics. *J. Cluster Computing*, 19(2):865–878, 2016.

[68] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, and M. Snir. Reducing waste in extreme scale systems through introspective analysis. In *IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 212–221, 2016.

[69] B. Nie, J. Xue, S. Gupta, C. Engelmann, E. Smirni, and D. Tiwari. Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities. In *IEEE 25th MASCOTS*, pages 22–31, 2017.

[70] N. Nakka, A. Agrawal, and A. Choudhary. Predicting node failure in high performance computing systems from failure and usage logs. In *IEEE Int. Symp. on Parallel and Distributed Process. Workshops and Phd Forum*, pages 1557–1566, May 2011.

[71] Li Yu, Zhou Zhou, Sean Wallace, Michael E. Papka, and Zhiling Lan. Quantitative modeling of power performance tradeoffs on extreme scale systems. *J. of Parallel and Distributed Computing*, 84(Supplement C):1 – 14, 2015.

[72] Sean Wallace, Zhou Zhou, Venkatram Vishwanath, Susan Coghlan, John Tramm, Zhiling Lan, and Michael E. Papka. Application power profiling on ibm blue gene/q. *Parallel Computing*, 57:73 – 86, 2016.

[73] Tao Li, Chunqiu Zeng, Yexi Jiang, Wubai Zhou, Liang Tang, Zheng Liu, and Yue Huang. Data-driven techniques in computing system management. *ACM Comput. Surv.*, 50(3):45:1–45:43, July 2017.

[74] Felix Salfner and Steffen Tschirpke. Error log processing for accurate failure prediction. In *Proceedings of the First USENIX conference on Analysis of system logs*. USENIX Association, 2008.

[75] Z. Zheng, Z. Lan, B. H. Park, and A. Geist. System log pre-processing to improve failure prediction. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 572–577, June 2009.

[76] B. C. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan. Logan: Problem diagnosis in the cloud using log-based reference models. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 62–67, April 2016.

[77] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1285–1298, 2017.

[78] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu. Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In *IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 447–455, June 2017.

[79] S. Kobayashi, K. Fukuda, and H. Esaki. Mining causes of network events in log data with causal inference. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 45–53, May 2017.

[80] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, pages 3111–3119, USA, 2013.

[81] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *CoRR*, abs/1402.3722, 2014.

[82] C. Bertero, M. Roy, C. Sauvanaud, and G. Tredan. Experience report: Log mining using natural language processing and application to anomaly detection. In *IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 351–360, 2017.

[83] Tech trends for 2015: the year in which the digital-first world takes hold. `http://www.techradar.com/us/news/world-of-tech/tech-trends-for-2015-the-year-in-which-the-digital-first-world-takes-hold-1280750`.

[84] Cisco adds big data to its 'internet of everything' push. `http://www.bloomberg.com/news/2014-12-11/cisco-adds-big-data-to-its-internet-of-everything-push.html`.

[85] Cisco global cloud index: Forecast and methodology, 2014-2019. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf`.

[86] M. Strohbach, H. Ziekow, V. Gazis, and N. Akiva. Towards a big data analytics framework for iot and smart city applications. *in Modeling and Processing for Next-Generation Big-Data Technologies: With Applications and Case Studies, Darmstadt, Springer pp*, pages 257–282, 2015.

[87] L. M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, October 2014.

[88] Levent Gurgen, Ozan Gunalp, Yazid Benazzouz, and Mathieu Gallissot. Self-aware cyber-physical systems and applications in smart buildings and cities. In *Proc. Conf. on Design, Automation and Test in Europe*, pages 1149–1154, San Jose, CA, USA, 2013. EDA Consortium.

[89] A. Costanzo, A. Faro, D. Giordano, and C. Spampinato. Implementing cyber physical social systems for smart cities: A semantic web perspective. In *13th IEEE Annual Consumer Communications Networking Conf. (CCNC)*, pages 274–275, Jan 2016.

[90] Andrea Giordano, Giandomenico Spezzano, and Andrea Vinci. Smart agents and fog computing for smart city applications. In *Proceedings of the First International Conference on Smart Cities - Volume 9704*, Smart-CT, pages 137–146, Berlin, Heidelberg, 2016. Springer-Verlag.

[91] Oliver Niggemann, Gautam Biswas, John S Kinnebrew, Hamed Khorasgani, Sören Volgmann, and Andreas Bunte. Data-driven monitoring of cyber-physical systems leveraging on big data and the internet-of-things for diagnosis and control. In *Proc. 26th Int. Workshop on Principles of Diagnosis*, pages 185–192, 2015.

[92] M. Yang, Y. Li, D. Jin, L. Zeng, X. Wu, and A. V. Vasilakos. Software-defined and virtualized future mobile and wireless networks: A survey. *Mobile Networks and Applications*, 20:4–18, 2015.

[93] T. Lin, J. m. Kang, H. Bannazadeh, and A. Leon-Garcia. *Enabling SDN applications on Software-Defined Infrastructure.* in IEEE Network Operations and Management Symposium (NOMS), Krakow, 2014.

[94] L. Bennacer, L. Ciavaglia, S. Ghamri-Doudane, A. Chibani, Y. Amirat, and A. Mellouk. *Scalable and fast root cause analysis using inter cluster inference.* in IEEE ICC - Next-Generation Networking Symposium, 2013.

[95] J. Kwisthout. Most probable explanations in Bayesian networks: Complexity and tractability. *International Journal of Approximate Reasoning*, 52(9):1452–1469, December 2011.

[96] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[97] Judea Pearl. Probabilistic reasoning in intelligent systems: Networks of plausible inference. *Synthese-Dordrecht*, 104(1):161, 1995.

[98] Todd Underwood. The death of system administration. *; login:: the magazine of USENIX & SAGE*, 39(2):6–8, 2014.

[99] Donghwoon Kwon, Hyunjoo Kim, Jinoh Kim, Sang C Suh, Ikkyun Kim, and Kuinam J Kim. A survey of deep learning-based network anomaly detection. *Cluster Computing*, pages 1–13, 2017.

[100] Sarah M. Erfani, Sutharshan Rajasegarar, Shanika Karunasekera, and Christopher Leckie. High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning. *Pattern Recognition*, 58:121 – 134, 2016.

[101] Sébastien Sorlin and Christine Solnon. Reactive tabu search for measuring graph similarity. In *Proceedings of the 5th IAPR International Conference on Graph-Based Representations in Pattern Recognition*, GbRPR, pages 172–182, Berlin, Heidelberg, 2005. Springer-Verlag.

[102] Zhibiao Wu and Martha Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics, 1994.

[103] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 302–308, 2014.

[104] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In Divyakant Agrawal, K. Selçuk Candan, and Wen-Syan Li, editors, *New Frontiers in Information and Software as Services*, pages 209–228, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[105] Soheil Danesh, Tamara Sumner, and James H Martin. Sgrank: Combining statistical and graphical methods to improve the state of the art in unsupervised keyphrase extraction. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics*, pages 117–126, 2015.

[106] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[107] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference*, USENIXATC, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[108] Marc Casas and Greg Bronevetsky. Prediction of the impact of network switch utilization on application performance via active measurement. *Parallel Comput.*, 67(Supplement C):38 – 56, 2017.

[109] Seren Soner and Can Azturan. Integer programming based heterogeneous cpu gpu cluster schedulers for slurm resource manager. *J. of Comput. and System Sciences*, 81(1):38 – 56, 2015.

[110] H. Subramoni, D. Bureddy, K. Kandalla, K. Schulz, B. Barth, J. Perkins, M. Arnold, and D. K. Panda. Design of network topology aware scheduling services for large infiniband clusters. In *2013 IEEE Int. Conf. on Cluster Comput. (CLUSTER)*, pages 1–8, 2013.

[111] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. Topology-aware resource management for hpc applications. In *Proc. of the 18th Int. Conf. on Distributed Comput. and Networking*, pages 17:1–17:10, 2017.

[112] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Trans. Syst., Man, Cybern., Syst.*, 21(3):660–674, 1991.

[113] Simon Haykin. *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994.

[114] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[115] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. of Mach. Learning Res.*, 15(1):1929–1958, 2014.

This thesis is supported by the BigStorage project (ref. 642963) funded by Marie Skłodowska-Curie ITN for Early Stage Researchers.