POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
CORSO DI LAUREA MAGISTRALE IN COMPUTER SCIENCE AND ENGINEERING

BEHOLDER: ONLINE MONITORING FRAMEWORK FOR ADAPTIVE APPLICATIONS

M.Sc Thesis of:
Alberto Bendin

Advisor:
Prof. Gianluca Palermo

Co-Advisors:
PhD. Davide Gadioli

Academic year 2017/2018

Ringrazio il mio Advisor Gianluca Palermo, e, soprattutto, il mio Co-Advisor Davide Gadioli, senza l'aiuto dei quali questo lavoro non sarebbe stato possibile.

Alla mia famiglia
e a tutti quelli che mi hanno sostenuto

Abstract

HIGH Performance Computing (HPC) allows scientists and engineers to solve complex computation-intensive problems of disparate nature.

HPC applications typically manage huge input datasets and are characterized by multiple parameters that influence their execution. The power consumed or dissipated limits modern systems performance. Approximate computing is an appealing approach to improve energy efficiency, leveraging the performance-accuracy trade-off. Autotuner frameworks are leading players in the actuation of this approach. They drive the target application by selecting the best set of application parameters according to user-set goals, expressed with respect to metrics of interest, such as execution time or power consumption. In other words, autotuners choose the best trade-off among the metrics which still satisfies the requirements. A key component of an autotuning framework is the application knowledge, which describes the application expected behavior. The autotuning framework learns this knowledge either at design time or at run-time. We will focus on the latter version.

In the occurrence of a change in the application behavior, be it following a variation in the input dataset features or an alteration in the execution environment, the knowledge with which the autotuner operates becomes obsolete. This impacts negatively on the application's efficiency. Thus emerges the need to cope with such situations. In order to do so, we need to solve a problem which is composed of (i) detecting consistent discrepancies between the expected and the observed application behavior, (ii) and adapting the application knowledge accordingly. The main objective of this work is the implementation of a module to continuously monitor HPC applications distributed on multiple nodes, together with its integration in a preexisting autotuner. This new remote module is in charge of monitoring the execution and detecting anomalies by means of a two-level hierarchical Change Detection Test (CDT). The first level, an ICI-based CDT, is in charge of the anomaly detection task by evaluating the collective behavior of the running nodes. The second level is a hypothesis test which works client-wise; it gauges the percentage of misbehaving nodes out of all the available and according to this it either confirms or rejects the change detected by the first level. We validated the proposed methodology on synthetic applications and on two case studies.

Sommario

IL calcolo ad elevate prestazioni (High Performance Computing - HPC) consente a scienziati e ingegneri di risolvere problemi complessi, di svariata natura, che richiedono molti calcoli.
Le applicazioni HPC tipicamente gestiscono enormi insiemi di dati in ingresso e sono caratterizzate da molteplici parametri che ne influenzano l'esecuzione. La potenza consumata o dissipata limita le performance dei sistemi moderni. Tecniche di calcolo approssimato che si basano su compromessi tra prestazioni e accuratezza sono interessanti approcci per migliorare l'efficienza energetica.
Gli Autotuner sono protagonisti nell'implementazione di queste strategie. Gestiscono l'applicazione interessata selezionando la migliore configurazione di parametri, secondo obiettivi impostati dall'utente rispetto a metriche di interesse, come tempo di esecuzione o consumo energetico. In altre parole, un autotuner sceglie il miglior compromesso tra le metriche che soddisfa i requisiti. Componente chiave di un sistema di autotuning è una conoscenza dell'applicazione che ne descrive il comportamento atteso. Gli autotuner ricevono questa conoscenza o prima dell'esecuzione o durante la stessa.

Nel caso di un cambiamento nel comportamento dell'applicazione, sia esso in seguito ad una variazione nelle caratteristiche dei dati in ingresso o ad un'alterazione nell'ambiente di esecuzione, la conoscenza con cui l'autotuner opera diventa obsoleta. Questo impatta negativamente sull'efficienza dell'applicazione. Emerge quindi la necessità di far fronte a queste evenienze. Per fare ciò, dobbiamo risolvere un problema composto dal (i) rilevare questa significativa discrepanza tra il comportamento atteso dell'applicazione e quello osservato, (ii) e adattare di conseguenza la conoscenza sull'applicazione. Il principale obiettivo di questo lavoro è l'implementazione di un modulo preposto al monitoraggio continuo di applicazioni HPC distribuite su più nodi, insieme alla sua integrazione in un preesistente autotuner. Tale nuovo modulo remoto è incaricato di monitorare l'andamento dell'esecuzione e di identificare anomalie per mezzo di un test di rilevamento di cambiamenti (Change Detection Test - CDT) gerarchico composto da due livelli. Il primo, un CDT basato sull'intersezione di intervalli di confidenza, è incaricato di rilevare l'irregolarità valutando il comportamento collettivo dei nodi in esecuzione. Il secondo livello è un test d'ipotesi che lavora su ogni singolo nodo;

stima la percentuale dei client che deviano dal comportamento atteso e a seconda di questa conferma o rigetta l'anomalia rilevata dal primo livello. Abbiamo validato la metodologia proposta su applicazioni sintetiche e su due casi di studio.

Contents

List of Figures

List of Algorithms

List of Acronyms

AS-RTM	Application-Specific RunTime Manager.
CDT	Change Detection Test.
CI	Confidence Interval.
CLT	Central Limit Theorem.
CPU	Central Processing Unit.
CQL	Cassandra Query Language.
CUSUM	CUmulative SUM.
CV	Coefficient of Variation.
DBMS	DataBase Management System.
DD	Detection Delay.
DoE	Design of Experiments.
DSE	Design Space Exploration.
FLOPS	Floating-Point Operations Per Second.
FN	False Negative.
FP	False Positive.
FPS	Frames Per Second.
FSA	Finite-State Automaton.
GPGPU	General-Purpose computing on Graphics Processing Units.
GPU	Graphics Processing Unit.

HPC	High Performance Computing.
ICI	Intersection of Confidence Intervals.
IDS	Intrusion Detection System.
IoT	Internet of Things.
LWT	Last Will and Testament.
M2M	Machine to Machine.
MAPE	Monitor-Analyze-Plan-Execute.
MQTT	Message Queuing Telemetry Transport.
OP	Operating Point.
PDF	Probability Density Function.
PU	Processing Unit.
QoS	Quality of Service.
RD	Recognition Delay.
RSD	Relative Standard Deviation.

CHAPTER 1

Introduction

Nowadays, increasingly accurate climate forecasts, biomedical research, business analytics, high-frequency trading and in general, Big Data problems, are examples of applications that require huge computing performance in order to obtain significant results. For these reasons, High Performance Computing (HPC) technologies have been continuously refined with the unique objective of achieving relevant speedups. However, while the architecture power rises always more, so does its energy consumption, until it hits a hard wall, caused, among other reasons, by thermal issues, such that heat dissipation capability becomes a power cap; the cooling of data centers is itself an ongoing concern nowadays, and there exist studies on how to cope with this issue without causing any harm to the environment [8] [19]. Energy consumption of data centers has become, obviously, a leading day-to-day matter too, and techniques to improve efficiency are continuously investigated; just to give a rough idea of how fast this problem is growing, the data centers in the US only are expected to consume 140 billion kWh in 2020, from 61 billion kWh in 2006 and 91 billion kWh in 2013 [1]. A proof of the attempt by the industry, to embark on a change of course towards more resource-consumption-aware policies, is the ranking by energy efficiency of the world's top 500 green supercomputers offered in [9].

A new trend in the design of HPC architectures is to rely, not only on homogeneous platforms, but also on heterogeneous ones; these follow an "offload" programming model, where an accelerator is programmed as a coprocessor, to speed up the execution of computationally intensive kernels. This ever-increasing number of processing units, integrated on the same many-core chip, delivers computational power that can exceed the performance requirements of a single application. The number of chips (and related power consumption) can thus be reduced to serve multiple applications - a practice which is called resource consolidation. However, this approach requires techniques to partition and assign resources among the applications and to manage unpredictable dynamic workloads; therefore, run-time adaptability represents a key requirement for computing systems to adjust their behavior with respect to operating environments, usage contexts, resource availability and even to faults, thus enabling close-to-optimal operation in the face of changing conditions [63].

Typically, HPC applications, being very complex in nature and computation-intensive, expose many parameters, known as software-knobs, which directly impact on the observed performance metrics, for instance, execution time or power consumption. This design allows an external agent to tweak these parameters in an attempt to maximize the metric(s) of interest according to the context and to the requirements. The high number of exposed parameters, together with the fact that often the application performance depends on many external factors, make the process of selecting the best knobs extremely complex.

The main purpose of an *Autotuner*, which is the core topic at the base of this work, is to solve by inspection a multi-objective constrained optimization problem that represents the application requirements. In other words, the autotuner automatically configures the application in the best possible way with respect to the user-selected goals, which can be on power-consumption, time constraints, or other performance metrics. There exist two approaches to the autotuning strategy: static and dynamic. The former focuses on exploring huge design spaces; its peculiarity is that the tuning is carried out only once at design time before the start of the application execution. Dynamic autotuners instead focus on adapting the application to run at its best; they allow for regular sensing of the current context to adapt to possible run-time requirements changes. Among the approaches that aim at improving the computation efficiency, approximate computing is an appealing path. A requirement for an application to be driven and controlled by autotuners which leverage approximate computing is that the nature of the application itself is compatible with the concept of approximation. Approximate computing is a technique which returns a possibly inaccurate result, rather than a guaranteed accurate result, in return for a gain in other metrics of interest, like a reduced execution time; if the application can tolerate a degradation in output quality, it can trade performance for accuracy. Previous results [57] show how a small reduction of the quality may lead to a significant speedup. Then, the definition of best configuration depends on the end-user requirements. Moreover, the autotuner has to own a knowledge about the target application, in the form of a model, which allows it to consistently drive the application, i.e. the observed performance reached by the related parameters configuration. Very strong assumption is the correctness of such application knowledge, which is critical for the success of the autotuning process since the autotuner relies entirely on that. Most autotuner frameworks do not have mechanisms to adapt to an inconsistent application knowledge, or their approach suffers from limitations in the contexts on which they can be applied.

This poses a new challenge, which is finding a way to consistently check for the application model robustness in the event of a possibly changing context or execution environment, thus allowing to take corrective actions. In the event of a change of any kind, the knowledge used by the autotuner to drive the application can be potentially invalidated. Be it for an unexpected change in the data on which the application is working, or for hardware faults which change the execution environment, or again for a power throttling following a trivial thermal alarm. If such irregularities remain undetected for a long period of time they can dangerously undermine the overall system efficiency and lead to higher costs or other forms of penalty; thus the need of a constant check of the correctness of the knowledge used by the autotuner.

What we propose with this work is an attempt at relaxing some key assumptions on the correctness of the application model used by the autotuner, by implementing a method to monitor such model at run-time and rendering possible its update, if any anomaly is detected. We based our work on a preexisting dynamic autotuner framework, mARGOt. Integrated with mARGOt is Agorà, a module which takes care of automatically profiling the application that must be tuned, in order to extract the knowledge that allows the autotuning. Peculiarity of Agorà is its ability to drive an efficient Design Space Exploration in a distributed fashion, fully exploiting in parallel the available nodes which run the same application. Moreover, Agorà is able to manage multiple applications simultaneously. It is important to notice that this modeling phase is carried out once and for all as soon as a new unknown application is integrated into the framework. We started from this, and created a new module for mARGOt, the "Beholder", which pushes to the next level the "automatism" and proactivity of the framework, by adding the ability to detect anomalies with respect to the expected behavior of the applications under mARGOt's control, in order to automatically re-learn the application knowledge without the need of any human interaction and, possibly, to inform the user of irregularities.

We propose a strategy based on a hierarchical Change Detection Test (CDT). The first level, implemented using an ICI-based CDT, evaluates the collective behavior of all the nodes running the same application, thus using the same model, to gauge the weight of possible anomalies with respect to the total number of clients. The second level consists of a hypothesis test carried out client-wise; it serves the purpose of assessing how many clients behave irregularly out of all the nodes running the same application. This is because the model retraining phase is expensive and should not be performed when just few nodes misbehave, otherwise it could lead to a loss of efficiency, above all in our field of application. According to this, the second level either rejects or confirms the detection of the first level. When an anomaly is confirmed by both the tests, we trigger a re-training request, which instructs Agorà to learn from scratch the application knowledge. A strong assumption of the proposed methodology is in the data monitored and used by the hierarchical CDT. It is required that the collected samples come from random variables which are independent and identically distributed.

We tested our approach on synthetic applications and on two case studies. In particular, we used a synthetic application to perform an evaluation of the parameters which control the CDT performance, with the objective to find a good compromise between the rate of false positives and the delay in the detection of a change. Then we tested the proposed enhanced framework with the K-means clustering application to prove the functioning of the methodology; in this experiment we concentrated on the first level of the hierarchical test. We introduced a power throttling by lowering the CPU frequency to simulate a real-world scenario where a thermal alarm could trigger a power-capping. The second case study is represented by the Stereomatch application. In this case, we introduced a change in the input data and concentrated the analysis on the second level of the hierarchical CDT; we demonstrated how significant the i.i.d assumption underlying the proposed methodology is, which is a requirement for both the application itself and the architecture.

1.1 Contribution

Since the application model is the ground truth with which an autotuner carries out its application-drive role, we propose a strategy to evaluate its correctness by analyzing the collective behavior of the running autotuned nodes, by means of what are typically referred to as Change Detection Tests (CDTs), which are methods designed to detect changes in data streams.

We introduce a new module in charge of performing a CDT, in the preexisting autotuner framework mARGOt, to increase the proactivity of the latter with respect to anomalies in the monitored processes. What we achieve with this new functionality is the ability to react to changes in the monitored application behavior, by updating the knowledge used by the autotuner to drive the application itself. Once aware of such irregularity, we can re-train the application model to adapt to the new context.

This thesis introduces multiple contributions, the main of these is the implementation of a hierarchical Change Detection Test strategy to discover anomalies in the monitored process and the modification of the baseline autotuner framework mARGOt to allow the re-learning of the application model during the production phase. The CDT is composed of two levels, the first is an ICI-based Change Detection Test, while the second is a hypothesis test to confirm or reject the detection of the first level. This required the introduction of mechanisms to automatically control the monitoring of computationally intensive and not always available metrics. For instance, the accuracy measure, i.e. the error with respect to the non-approximate computation, is only computed during the training phase, and has to be automatically disabled during the production phase. We evaluated the proposed methodology using synthetic applications and two case studies.

1.2 Organization of the Thesis

This thesis is structured as follows: Chapter 2 addresses the current state-of-the-art of the autotuning and change detection techniques, given that those are the two underlying topics at the base of this thesis. In Chapter 3 we present the structure of the preexisting autotuner framework which is the starting point for this work, the architecture we target and the MQTT communication protocol. Chapter 4 addresses the proposed methodology and all the design choices made while engineering the Beholder module, like the interactions with the other modules and the double hierarchical level which represent the change detection test. Chapter 5 contains all the technical implementation details of the Beholder module, with the overall framework workflow and behavior. In Chapter 6 we propose the experimental results carried out both with synthetic and real applications to validate the proposed approach and prove its benefits. Chapter 7 summarizes the conclusions we drew from this work and presents future works.

CHAPTER 2

State-of-the-Art

In this chapter we discuss about the current state-of-the-art of the subjects on which this work is based. This thesis proposes the integration of an online change detection mechanism, a statistical analysis technique employed in many fields, in an autotuner framework. The objective is to increase the proactivity of the latter in the management of the application, especially for those situations in which a consistent change invalidates the knowledge used by the autotuner to drive the application. We provide awareness of such cases to the system, which is then able to take corrective actions, typically involving a retraining of the application model to avoid loss of efficiency or other forms of penalty.

Since the main contribution of this thesis is the union of the application autotuning and change detection domains, which, to the best of our knowledge, has never been proposed before, we approach the state-of-the-art by organizing the discussion in this way: first we introduce the autotuning subject and go through techniques of autotuning, explaining the reason behind the choice of mARGOt as target autotuner on which to work, then we describe the change detection problem in a generalized fashion while also illustrating the algorithm we will use.

2.1 Tunable applications

Several classes of applications expose application-specific parameters, also known as dynamic knobs [45], that influence the algorithm behavior, such as the resolution in a video processing application or the number of trials in a Monte-Carlo solver. Writing applications in a parametric way ensures better maintainability and portability and as such is a good practice. A change in the values of those parameters can influence the performance of the application, because they can be related to the amount of data to be processed or iterations to be done for completing the assigned task. Typically, approximate computing techniques are used to trade off accuracy of results for performance metrics, such as latency or energy consumption. Modern applications are growing in complexity and they usually expose an increasing number of algorithm-specific parameters and relevant target metrics. The selection of the optimal application configuration is a difficult and time-consuming task, since the design space might be huge and the performance might be composed of conflicting metrics. This makes a manual selection of a single configuration, as output of the optimization process, very complex [38].

2.1.1 Intra-Functional and Extra-Functional Knobs

The dynamic knobs that are exposed for tuning can belong to different "layers", they can be either hardware-related or application-related.

2.1.1.1 Extra-Functional Knobs

Multi-core systems are the de facto standard in modern platforms and the trend is to further promote heterogeneity. One effect of using such architectures is that they introduce additional resource-related parameters that can be exposed at application level (for instance the number of threads) not directly impacting the functionality of the algorithm, while modifying its extra-functional characteristics [38].

2.1.1.2 Intra-Functional Knobs

Intra-Functional knobs directly impact the application leveraging approximate computing. Approximate computing can be used for applications where an approximate result is sufficient for its purpose. One example of such a situation is the occasional dropping of frames in a video application which can go undetected due to perceptual limitations of humans. Approximate computing is based on the observation that in many scenarios allowing bounded approximation can provide disproportionate gains in performance and energy, still achieving acceptable result accuracy. For example, in the K-means clustering algorithm, allowing only a 5% loss in classification accuracy can provide 50 times the energy saving compared to the fully accurate classification [59]. The most common approximation techniques are *Loop Perforation* and *Data-Type Precision*:

- Loop Perforation: involves the skipping of some loop iterations, thus reducing the overall computation complexity (and accuracy) to decrease execution time and energy consumption as well [58]. There are different techniques to determine the perforation rate, often depending on the distribution of the data on which the algorithm works [21].
- Data-Type Precision: leverages a loss of information in order to achieve a gain in performance, in the form of a switch between the data types used to store the data. For instance casting a `double` to a `float` or a `float` to an `int`.

2.2 Autotuning

A set of knob values describing the execution setting of a tunable application is known as *Configuration*:

$$configuration = \langle knob1,\ knob2,\ ...,\ knobN \rangle.$$

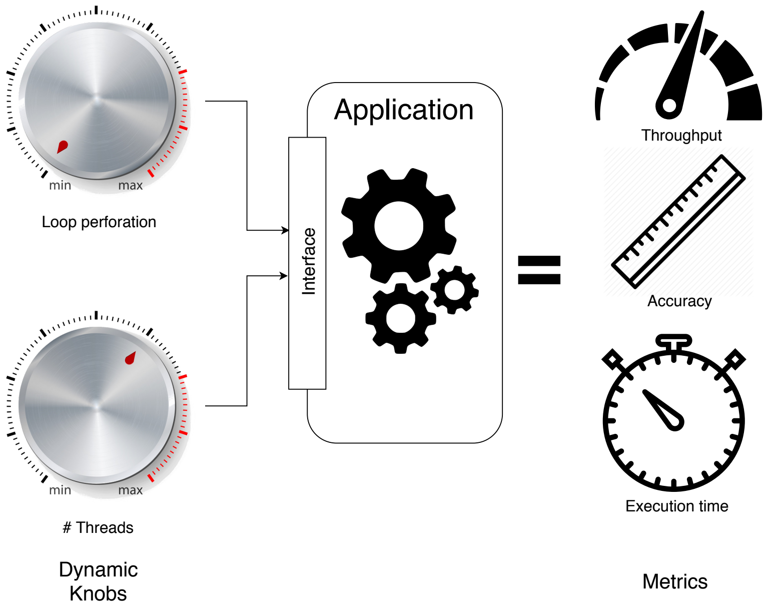


Figure 2.1: *Symbolic example of a tunable application which exposes two knobs, whose setting impact on three metrics of interest.*

Autotuning is defined as the task to automatically find the best configuration satisfying the application requirements. In general, a configuration is chosen as it is the solution of a particular mono or multi-objective user-specified optimization function. From now on, we will refer to a general performance parameter, such as execution time or energy consumption as a *Metric* (Figure 2.1). The application's goals and requirements are expressed with respect to its metrics. The metrics of interest of many applications often depend not only on their configuration but also on the features of their input data (if there is any), e.g. the input dataset size. The knowledge with which the autotuner manages the application, derived from a profiling phase of the latter, is represented by the set of rules that correlate the (possible) input features and a configuration with the corresponding observed metric(s) at which the application operates.

$$\langle [inputfeatures],\ configuration\rangle \implies \langle metric1,\ metric2,\ ...,\ metricN\rangle$$

Therefore we can rephrase the definition of autotuner as an application-driver which, once in possession of the target application's knowledge, manages this application by forcing it to run with the configuration which best satisfies the predefined requirements. Ultimately, an autotuner is just a selector of one configuration among many.

A common optimization problem is the maximization or minimization of a metric which may be in contrast with another one; this leads to a trade-off between the two. For instance, maximizing accuracy is in contrast with minimizing the execution time and energy consumption. For this reason, often every metric has a weight associated to it, so that the optimization problem can be solved taking into account the user-assigned importance of every objective.

2.2.1 Design-Time vs Dynamic Autotuning

Autotuning was mainly considered as a design-time phase to be carried-out offline, once. This meant that the knobs setting was chosen before starting the application, and that configuration stuck until the end of the execution. This is enough just in those circumstances in which the execution context is predictable and can be considered only a starting point, since applications might change their requirements at run-time, depending on external conditions and on the execution environment. Therefore, the pre-determined application configuration could be no longer appropriate. An example of this occurrence could be if the platform on which the application is running is mobile: when it operates on battery, the application might be interested in a more energy-aware configuration, but if the power adaptor is plugged in, it will change its requirements towards a more performance-oriented one.

Additional situations which can be regarded as sudden changes in the external conditions, to which the application has to react accordingly at run-time, are represented by events such as power throttling triggered by a thermal alarm for overheating. Typically, in these circumstances the CPU frequency is lowered, effectively impacting on the performance of the system and invalidating the application knowledge pre-computed when there was no power throttling in place. Moreover, a run-time change requiring a re-tuning can happen for those applications whose metrics are strongly dependent on the features of the input data, and a big change in the processed data could then invalidate the application knowledge (e.g. for Video Processing applications) [38]. These occurrences clearly show the need for applications to run-time adapt; thus a switch from a design-time autotuning to a dynamic one is required.

2.2.2 Dynamic Autotuning

Dynamic Autotuning can be therefore defined as the capability to find the best set of knob values, in an automatic and systematic way, that satisfies the application requirements at run-time, properly reacting to possible objective function changes. In order to adapt with respect to the actual situation, the autotuner must be able to gather insight on how the application is behaving. In order to receive this feedback on the actual application performance, the core interactions between the autotuner and the managed application are:

- reading the (user-set) application requirements;
- observing and monitoring of the actual behavior of the application with the current configuration of knobs;
- check whether the observed metrics are satisfying the requirements; if this is not the case then retrieve the most suitable configuration for the new situation and provide this updated configuration of knobs to the application;
- if the application is input-data dependent, take into account the current input features to be proactive in that respect.

This is practically put in place by wrapping with observer monitors the core regions of code that are managed by the autotuner, usually the body of the loops which include the most demanding and tunable kernels; ideally, the application should deploy a monitor for any metric of interest. The monitors can be of various nature, among others, there are time monitors to check the execution time, power monitors to observe the real-time energy consumption, error monitors to compute the accuracy with respect to the perfect computation without approximation.

2.2.3 Autotuning Related Works and Contribution

Especially in the context of HPC, there are several autotuning frameworks. Among these, many target specific tasks: ATLAS [72] for matrix multiplication routine, FTTW [37] for FFTs operations, are examples in this area. Because of their application-specific design, those types of approaches are not easily portable to other application fields.

Other frameworks are more independent from the application domain; examples of these are mARGOt [38] [39], the Green framework [29] and PowerDial [45]. In particular, the Green framework [29] uses loop and function approximation to perform a trade-off between a single Quality of Service (QoS) metric loss and execution time improvement. The problem of this approach is that it provides limited reaction capabilities (it provides a mechanism to sporadically update its information at run-time) and its constrained applicability to a limited set of kernels. In PowerDial, the authors propose a more general framework, that enables an application to adapt for reaching its performance goal; it uses the concept of application heartbeats [44] to monitor the run-time behavior. mARGOt and PowerDial address the same problem, however, mARGOt enables a developer to exploit trade-offs among more than two metrics and provides a generalized monitoring infrastructure to sense the application performance and execution environment; it is also able to take into account the input features.

Capri framework [69], instead, focuses on controlling tunable applications through the use of Approximate Computing [59]. There are two phases: an offline training which estimates the error and cost function using Machine Learning techniques; while the online control algorithm finds the best knobs setting that fulfills the objectives. Since Capri does not address stream applications, it is not investigating any reaction mechanism to adapt the application knowledge according to system evolution. Besides, due to the chosen formulation of the problem, the feasible region given by the error function does not depend on the actual input. This assumption might miss optimization opportunities.

Other autotuners allow for domain-specific techniques: ATune-IL [20] and PetaBricks [27] mainly rely on loop-related knobs in the code and predictable execution context; their peculiarity is that sometimes using a domain-informed search space representation is critical to achieving good results.

Since we propose a new module that will be integrated into an autotuner framework, we analyzed the state-of-the-art to carefully choose the target preexisting autotuner. Eventually, many reasons made mARGOt stand out as the best candidate framework for our work; some of these are that mARGOt has been designed to autotune any kind of software knobs exposed by the application, moreover mARGOt exploits its flexibility at a very limited integration cost in terms of additional lines of code and workflow refactoring. Furthermore, the mARGOT framework already comes with a learning module, Agorà, which coupled with the autotuner provides the latter with the model of the target application, thus automatically and seamlessly carrying out the task of application profiling needed to extract the knowledge that allows the autotuning itself. The operating mode of Agorà enables a totally new approach to the problem of application behavior profiling, since there is no need for an offline phase carried out aside and in advance. This radically transforms the typical workflow, allowing the user to run its application directly on the execution environment of the production phase. Besides, Agorà does not ground its workflow on predetermined objectives; it is completely agnostic about the application requirements. Furthermore, the computational power of the machine on which the tunable application is being executed is not kept busy by Agorà, which does the heavy job remotely on a dedicated server. This is a very appealing feature, enabler of the ultimate goal of the work we propose, that is to say, to retrain the application model if the observed application behavior deviates from it.

We propose the integration of a new module, in charge of the change point detection task, on top of a preexisting autotuner framework. Our objective is to increase the proactivity of the framework with respect to relevant changes of the observed application run-time behavior, to consequently force an update of the application model tailored for the new context. Our work goes in the direction of further relaxing underlying assumptions with respect to the limited adaptability of the framework in front of irregularities. Other works either do not offer any reaction policy, or they are based on trial and error approaches, or again they offload the task to other levels, or they require strong assumptions with respect to the use case.

There are frameworks which theoretically do not need the added proactivity we want to introduce in mARGOt, but still, they lack some features or they target different problems. Among these we can cite SiblingRivalry [28], which proposes an always online autotuning framework which eliminates the offline learning phase; it divides the available resources in half and selects two configurations, a "safe" one and an "experimental" one, according to an internal fitness function value. The online learner handles the current request in parallel on each of these two configurations and, according to the candidate that finishes first and meets the goals, it updates its internal knowledge about the configurations just used for the current objective function. This goes on until the context changes and, therefore, new objectives have to be achieved and then the procedure restarts all over again. SiblingRivalry does not need a change detection technique, that is the proposed contribution of our work, since it enforces a retraining for every context change, and the check for environment changes is continuous. This approach is different from ours, we are interested in updating the application model only when it is not sound anymore, and this is verified through the detection of a collective change in the behavior of the nodes running the application.

Among the previous works that manage the throughput-accuracy trade-off, the IRA framework [51] proposes an interesting approach to adapt an application at run-time. It investigates several features of the input, such as the mean value or its autocorrelation, to generate a canary input. The latter is the smallest subsampling of the actual input which has the same property as the original input. It uses a statistical hypothesis test to perform such an evaluation. However, the related paper investigates techniques for sub-sampling only images or matrix-like inputs. IRA uses the canary input to perform a Design Space Exploration (DSE) for each input, selecting as the most suitable software-knobs configuration the fastest one within a given bound on the minimum accuracy. Then it uses the best configuration with the actual input to produce the desired output. Our framework aims at having a wider applicability range, not just images or matrixes.

A rather different approach with respect to the previous ones is Anytime Automaton [55]. It suggests source code transformations to re-write the application using a pipeline design pattern. The idea is that the longer the given input executes in the pipeline, the more accurate the output becomes. The work targets hard constraint on the execution time, interrupting the algorithm when it depletes the time budget. In this way, it is possible to have guarantees on the feasible maximum accuracy. This is not a framework and it requires onerous modifications to the application source code.

2.3 Change Detection

In statistical analysis, change detection or change point detection tries to identify the time when the probability distribution of a stochastic process or time series changes. In general, the problem concerns both detecting whether or not a change has occurred, or whether several changes might have occurred, and identifying the time of any such changes. More generally change detection also includes the detection of anomalous behavior, in which an anomaly is the deviation in a quantity from its expected value, e.g., the difference between a measurement and a model prediction. Specific applications, like step detection and edge detection, may be concerned with changes in the mean, variance, correlation, or spectral density of the process.

Change Detection Tests (CDTs) are often used in manufacturing (quality control and fault detection), intrusion detection, spam filtering, website tracking, and medical diagnostics (system health monitoring), but also to spot bank frauds, structural defects or errors in a text. Reliable systems designed for processing data collected from real-world phenomena (e.g. for classification, sensing and monitoring purposes), have to handle the occurrence of unpredictable events to promptly adjust to the new operating conditions, in order to preserve performance [26], [22].

2.3.1 Online and Offline Change Detection

When the algorithms are run *online* as the data are coming in, especially with the aim of producing an alert, this is an application of sequential analysis where the sample size is not fixed in advance [32]. In statistics and signal processing, *Step detection* is the process of finding abrupt changes in the mean level of a time series or signal. One of the most popular streaming algorithms is the “sliding-window” one, which by considering a small “window” of the signal, looks for evidence of a step occurring within the window. The window “slides” across the time series, one-time step at a time.

By contrast, *offline* algorithms are applied to the data potentially long after it has been received. Offline approaches cannot be used on streaming data because they need to compare to statistics of the complete time series, and cannot react to changes in real time but often provide a more accurate estimation of the change time and magnitude.

2.3.2 Change Detection Challenges

Taking into consideration the sequential analysis approach, any change test must make a trade-off between these common metrics:

- False alarm rate;
- Misdetection rate;

- Detection delay.

Each of these metrics has to be taken into account and weighted for the specific application. For instance, considering an anti-spam filter, this translates in assessing the trade-offs between incorrectly rejecting legitimate email (false positives) as opposed to not rejecting all spam (false negatives) and the associated costs in time, effort, and cost of wrongfully obstructing good mail. Often, the change is small and the time series is corrupted by noise, and this makes the problem challenging because the step may be hidden by the noise, thus increasing the time-to-detection.

2.3.3 Applications

We present some real-world use cases where CDT techniques are employed, to show their effectiveness in a wide range of applications.

2.3.3.1 Edge Detection for Image Analysis

Edge detection is a fundamental tool in image processing, machine vision, and computer vision, particularly in the areas of feature detection and feature extraction [71]. Edge detection includes a variety of mathematical methods that aim at identifying points in a digital image at which the image brightness changes sharply or, more formally, has discontinuities. The points at which image brightness changes sharply are typically organized into a set of curved line segments knows as edges.

One of the many real-world applications of such a change detection is in the identification of abandoned objects from video surveillance, e.g. bags left in public places, which is crucial in terrorist attacks prevention [70]. The same goes for fully automatic systems in which alerts have to be triggered from intelligent video analysis, such as crowd counting, or virtual line-crossing. Another example of features extraction from image analysis is the detection of ecosystem disturbances through a comparison of satellite images of the same ground area taken in different periods. Again edge detection from aerial images is used for highway traffic incident detection [61] or possible illegal forest clearing discovery.

Automated change detection is prone to raise false alarms. For instance, differing weather conditions, the time of the year in which imagery is captured, and differences in acquisition angles can make buildings or scenery appear differently in images of the same location [65]. These factors have to be properly taken into account and compensated.

2.3.3.2 Intrusion Detection System

Intrusion Detection Systems (IDSs) is another field in which change detection is employed. IDS are devices or software applications that monitor a network or systems for malicious activity, resource misuse or policy violations. Anomaly-based Intrusion Detection System were primarily introduced to detect unknown attacks, in part due to the rapid development of malware. The basic approach is to use machine learning to create a model by monitoring trustworthy network traffic activity, and then compare new behaviors against this model [67]. Although this approach enables the detection of previously unknown attacks, it may suffer from false positives: previously unknown legitimate activity may also be classified as malicious. Obviously, also false negatives

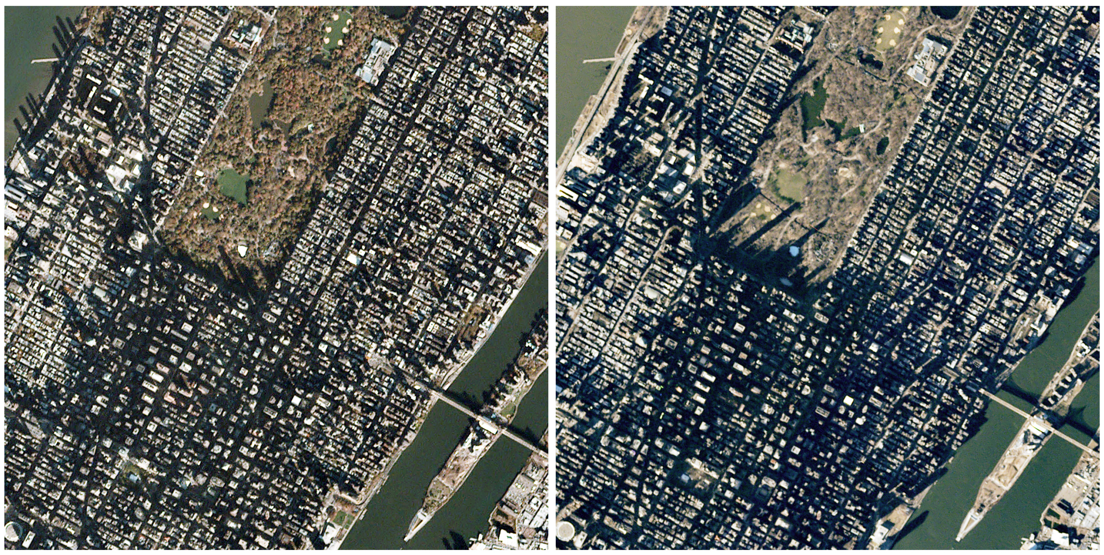


Figure 2.2: *Two satellite images of Manhattan taken in November 2016 (left) and February 2017 (right). Differences in sunlight conditions can create false positives of detected changes [65].*

could potentially occur, when a still unknown malicious behavior is used for training the established baseline which identifies what is "normal" for that network.

2.3.3.3 Sensors Networks

One last example is an application where change detection is used for fault detection purposes. Given a sensors network with redundant information, for instance, a network of sensors to monitor the temperature in a building with room-granularity, one can use the redundancy from having correlated readings from every zone to enforce a check on the sensors' behavior. For example, if a sensor observes a sudden increase in the reading, if the correlated sensors do not reflect the same step one can assume that the incoherent measure comes from a faulty sensor (it may just be that the sensor is near a heat source or it is flooded with sunlight). On the contrary, when all the three correlated sensors reflect the change then one can assume that the detected change has actually occurred [34].

2.3.4 CDT Related Works

The design of a test that can effectively detect changes in the stationarity of a process generating data is a challenging problem, in particular when the process is unknown, and the only information available has to be extracted from a set of observations [23]. There is a wide literature concerning change detection tests. The most common change detection techniques are parametric and non-parametric statistical tests [35], [31]. Parametric techniques require availability (or an estimate) of the probability density function of the process generating the data before and after the change and/or information about the nature of the drift itself. Conversely, non-parametric tests do not require strong a priori information and are thus more flexible in dealing with a large class of applications [33], [23].

The Mann-Whitney U test for independent samples [43] (which relies on the possibility to rank two independent samples of observations) is a nonparametric test originally designed for verifying whether two independent samples come from the same distribution or not. Its disadvantages are that it was not designed to work in sequential analysis and focuses on detecting a single change point.

Differently, the Mann-Kendall [48] (designed for environmental sciences) and the CUmulative SUM (CUSUM) [53] (developed in the system control community to detect structural changes) are non-parametric tests particularly suitable in sequential analysis. In minimax change detection, the objective is to minimize the expected detection delay for worst-case change-time distributions, subject to a cost or constraint on false alarms. A key technique for minimax change detection is the CUSUM procedure, which has also been successfully used in several diversified applications such as fault detection, onset detection in seismic signal processing, changes in mechanical systems.

Unfortunately, these tests require a design-time configuration phase to fix the test parameters (e.g., Mann-Kendall requires a significance level for the test while CUSUM needs to fix thresholds to detect changes); such parameters must be identified by exploiting a priori information or through a trial-and-error approach.

A different approach is presented in [25] where two automatic tests for detecting non-stationarity phenomena, which differ for detection accuracy and computational complexity, are suggested. These tests are particularly interesting since they propose a parameter configuration phase that allows for automatically configuring the parameters of the tests (i.e., they do neither require a priori information nor assumptions about the process generating the data) [23].

2.3.5 CDT using the ICI Rule

The work in [23] introduces a novel approach in designing non-parametric change detection tests that have to operate without any a priori information about the process generating data, as in [25]. Changes are detected by extracting features first and then applying the Intersection of Confidence Intervals (ICI) rule [42]. More specifically, the feature extraction step is meant to produce Gaussian distributed features from observations while the ICI rule is used to assess the stationarity of the features, and hence that of the data generating process. The ICI rule is very often used for signal denoising purposes [52], as shown in Figure 2.3 and it has proved to be particularly effective in image processing applications [46].

We chose to use the ICI CDT presented in [23] for our module, Beholder. The reason underlying the decision of this particular test is the flexibility derived from its non-necessity of a priori information on the observed process; besides, being non-parametric, its applicability is much wider than any parametric method. Also, due to the reliance on fewer assumptions, non-parametric methods are more robust and paradoxically simpler to use and less prone to errors.

2.3.5.1 Problem Statement

Let $X : \mathbb{N} \to \mathbb{R}^d$ be a stochastic process generating independent and identically distributed (i.i.d.) d dimensional real data (observations) according to an unknown Probability Density Function (PDF). Let $O_T = \{X(t),\ t = 1, \dots, T\}$ be the sequence of observations measured up to time T, and assume that at least the first $T_0 < T$

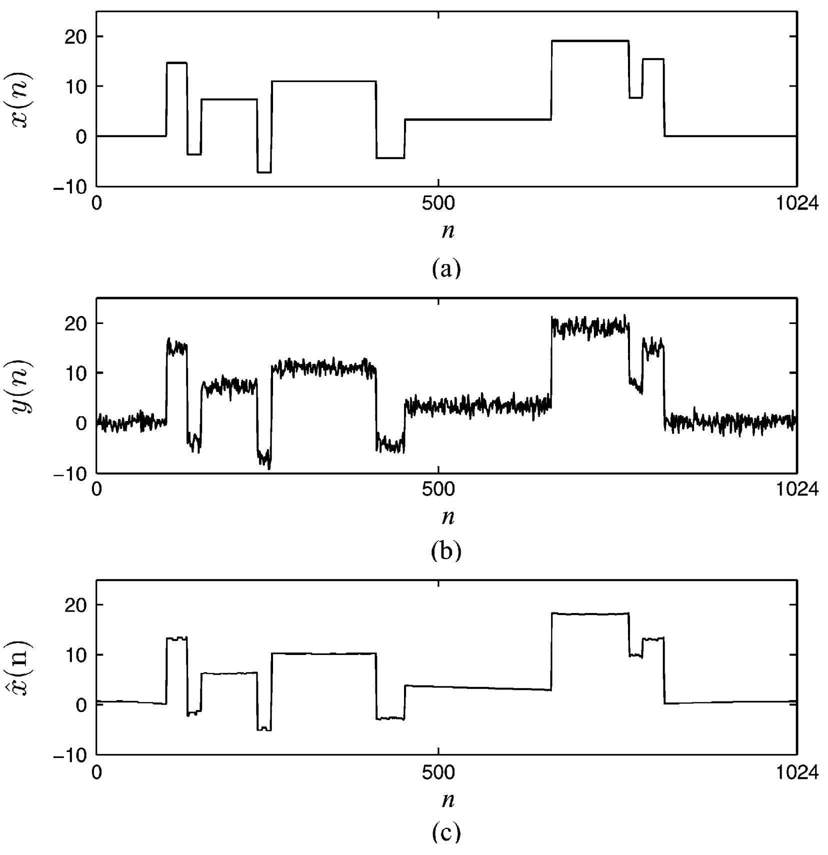


Figure 2.3: *Denoising algorithm which uses Local Polynomial Approximation (LPA) [52] employing the ICI rule. Blocks signal: (a) Noise free. (b) Noisy. (c) Signal denoised by the LPA-ICI algorithm. Picture taken from [52]*

observations come from the process X in a stationary state. As such, $X(t)$ is a d-dimensional vector of measurements at time t. Define $TS = O_{T_0}$ the set of initial supervised observations that constitute the training sequence of the change detection test.

The goal of a change detection test is to determine the time instant $t = T^*$ in which the process X changes its statistical properties. Since this work is meant for application scenarios where no a priori information is available (i.e., we are not requiring the knowledge of the distribution of the process generating the data), the change detection test solely relies on features extracted from the observations.

2.3.5.2 ICI Rule

The ICI rule operates, combined with a polynomial regression technique, on sequences of noisy data $z \in \mathbb{R}$ extracted with a given sampling frequency from a Gaussian distribution of mean $\mu(t)$ and standard deviation σ

$$z(t) \sim \mathcal{N}(\mu(t),\ \sigma^2)\ \ t \in W, \tag{2.1}$$

where t represents the time instant and W the uniformly spaced sampling grid. For each $t \in W$, the ICI rule adaptively identifies an optimal neighborhood $U_{i^+}(t)$ and substitutes (regularizes) $z(t)$ with an estimate $\hat{\mu}(t)$ of $\mu(t)$, obtained by least squared

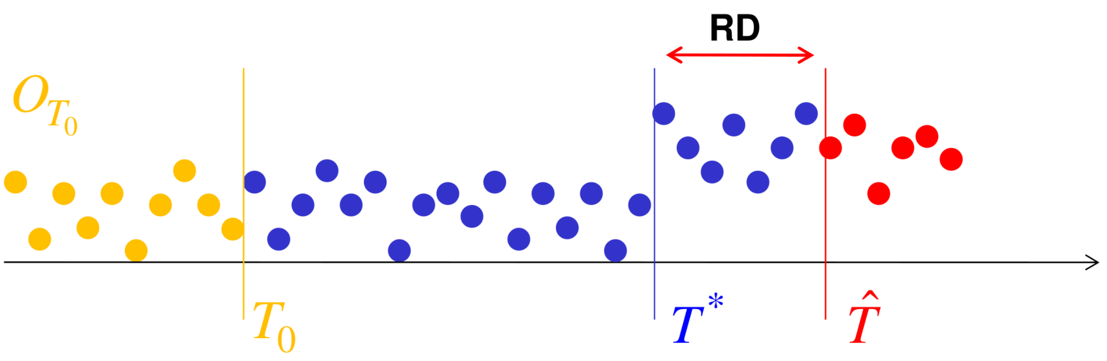


Figure 2.4: *Example of observations coming from a process subject to a change at time T^*. The variation is only detected at time $\hat{T}$, thus the $RD = \hat{T} - T^*$. T_0 is the last observation belonging to the training set used to configure the test. Picture taken from [41].*

error polynomial fit of order m of the noisy data belonging to the optimal neighborhood $\{z(t),\ t \in U_{i^+}(t)\}$. The neighborhood $U_{i^+}(t)$ is adaptively selected from a set of L nested neighborhoods $\{U_i(t)\ \ i = 1, \ldots, L\}$, obtained by scaling a reference neighborhood $\tilde{U}(t)$ centered at t. Let $\{\hat{\mu}_i(t)\ \ i = 1, \ldots, L\}$ be the sequence of estimates $\hat{\mu}(t)$ evaluated over $U_i(t)$, and $\{\sigma_i,\ i = 1, \ldots, L\}$ be the associated standard deviations. Note that since σ is constant in (2.1), σ_i depends only on $U_i(t)$, σ and m.

The ICI rule can be stated as follows. Let $\mathcal{I}_i$ be the confidence interval of $\hat{\mu}_i(t)$

$$\mathcal{I}_i = [\hat{\mu}_i(t) - \Gamma\sigma_i(t);\ \hat{\mu}_i(t) + \Gamma\sigma_i(t)], \tag{2.2}$$

where $\Gamma > 0$ is a parameter configured at design time to balance the change detection performance evaluated on the number of False Positives (False Positives (FPs)) and False Negatives (FNs); it also directly impacts the Recognition Delay (RD) (also known as Detection Delay (DD)), which is evident in Figure 2.4. The ICI rule selects the adaptive neighborhood $U_{i^+}(t)$ as the one corresponding to the largest index j for which $\bigcap_{i=1}^{j} \mathcal{I}_i$ is not empty [23].

2.3.5.3 CDT using the ICI rule

The test relies on a set of functions $\{F^k\ k = 1, \ldots, N\}$ that transform the observations $A \subset O_T$ into values $F^k(A) \in \mathbb{R}^d$, which are called features. When the features are distributed as in (2.1), we can use U_{i^+}, the ICI-selected interval as a valuable tool for change detection. Stationarity in the process generating the data is monitored by using each feature separately: a change is detected in X when at least one of the features shows a change. As such, we can focus on a single feature and devise change detection tests using the ICI rule on a single feature output sequence. Given that we want to detect a change in the features' distribution we employ the ICI rule to select the adaptive neighborhood among intervals having the leftmost extreme fixed at $t = 1$, i.e. $[1, T]$, $T > 0$ (Figure 2.6b). Thus, the set of neighborhoods considered for analyzing observations at time T is $\{[1, T_0], \ldots, [1, T]\}$.

Finally, the estimates associated to each neighborhood are obtained by the 0^{th} order polynomial fit of feature values: this choice reflects the intuitive idea that stationary processes provide constant feature values. Any ideal neighborhood selected by the

ICI rule contains a set of features that must be considered constant in a stationarity scenario and have been generated from the same Gaussian distribution. Contrarily, feature values belonging to different ICI-selected neighborhoods should be considered as values generated by different Gaussian distributions: Figure 2.6 illustrates a complete example of change detection selection using the ICI rule.

Let $\{F^k(t)\ t = 1, \ldots, T_0\}$ be the sequence of values assumed by the feature F^k within the training set; then

$$\hat{\mu}^k_{T_0} = \sum_{t=1}^{T_0} \frac{F^k(t)}{T_0} \quad \text{and} \quad \hat{\sigma}^k_{T_0} = \sqrt{\sum_{t=1}^{T_0} \frac{(F^k(t) - \hat{\mu}^k_{T_0})^2}{T_0 - 1}} \tag{2.3}$$

represent an estimate of the expected value of the k^{th} feature and its standard deviation, when X is in the initial stationary state. Note that (2.3) defines the first confidence interval $\mathcal{I}^k_{T_0}$ according to (2.2), and thus the test configuration consists of computing $\mathcal{I}^k_{T_0}$, $\hat{\mu}^k_{T_0}$ and $\hat{\sigma}^k_{T_0}$.

During the operational phase, for each $T > T_0$, the test computes the 0^{th} order polynomial fit of the features belonging to the interval $[0, T]$. Then, the ICI rule is used to assess whether these values have been generated from the same Gaussian distribution or not. Hence, the ICI rule acts as a nonstationarity test determining, by means of the features, if X is constant within the time interval $[0, T]$, and thus stationary. The estimate coming from the 0^{th} order polynomial fit and its standard deviation can be efficiently computed by means of the following iterative formulas:

$$\hat{\mu}^k_T = \frac{(T-1)\mu^k_T + F^k(T)}{T} \quad \text{and} \quad \hat{\sigma}^k_T = \frac{\hat{\sigma}^k_{T_0}}{\sqrt{T}}\ ,\ \forall T > T_0. \tag{2.4}$$

Finally, the ICI rule reveals a change in stationarity in X when the $\bigcap_{t=T_0}^{T} \mathcal{I}^k_t$, $T > T_0$ becomes an empty set [23].

2.3.5.4 Final CDT algorithm

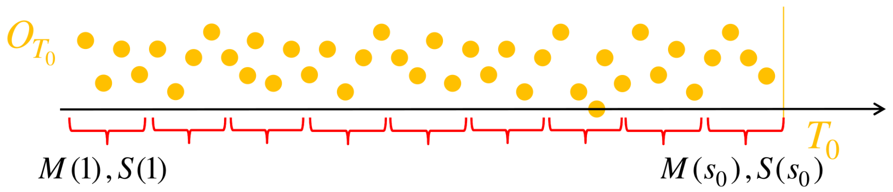
The set of observations $X(t)$ is segmented in disjoint subsequences of length ν:

$$Y_s = \{X(t), (s-1)\nu \le t < s\nu\}, \tag{2.5}$$

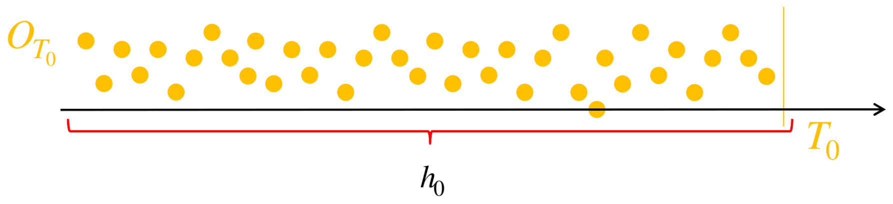
where $s \in \mathbb{N}^*$ is the subsequence index. The test exploits features that are evaluated only on these subsequences: changes in X are detected as events occurring within a subsequence, and thus with an accuracy $\pm\nu/2$. To simplify the notation used, the following abbreviation is introduced: $F^k(s) = F^k(Y_s)$, $k = 1, \ldots, n$. The polynomial fit of 0^{th} order and the ICI rule are then applied to the set of subsequences indexes $\{[1, S_0], \ldots, [1, T/\nu]\}$, being $S_0 = T_0/\nu$. The implemented test relies on two features only, computed on each subsequence. The first feature is the sample mean M evaluated in each subsequence Y_s:

$$M(s) = \frac{1}{\nu} \sum_{t=(s-1)\nu}^{s\nu} X(t). \tag{2.6}$$

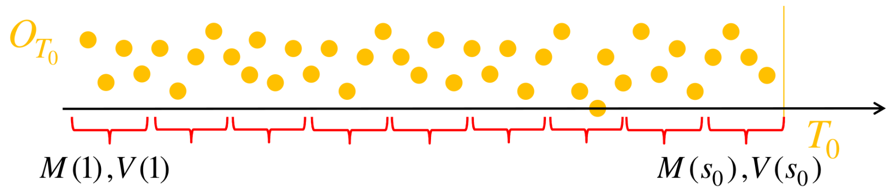
Since $X(t)$s are i.i.d. (when X is stationary), the Central Limit Theorem guarantees that $M(s)$ approaches a Gaussian distribution (and thus these values are distributed



(a) *The training set O_{T_0} is divided into non-overlapping subsequences of ν observations.*



(b) *The whole training set is used to compute the exponent h_0.*



(c) *The values of the sample variance are replaced by $V(s) = \mathcal{T}(S(s))$.*

Figure 2.5: *Training phase used for configuring the ICI-based CDT. Picture taken from [41].*

as in Equation 2.1). The second feature is a quadratic deviation, evaluated in each subsequence Y_s:

$$S(s) = \sum_{t=(s-1)\nu}^{s\nu} (X(t) - M(s))^2. \tag{2.7}$$

Note that $S(s)/(\nu - 1)$ is the sample variance of Y_s. Unfortunately, $S(s)$ is not Gaussian distributed when the samples $X(t)$ are drawn from an arbitrary distribution. Thus, the power-law transform presented in [62] is adopted to obtain a derived feature, which is related to the variance of X and is drawn from a Gaussian distribution. This Gaussian approximation holds when the sample variance is computed from i.i.d. samples; the transform has the form

$$\mathcal{T}(S(s)) = \left(\frac{S(s)}{\nu - 1}\right)^{h_0}, \tag{2.8}$$

where the exponent h_0 is expressed as a function of the cumulants of X, which in turn can be estimated from the observations (refer to [62] for details about this transform). Algorithm 2 explains how the cumulants of X, and hence h_0, can be computed from the observations in the training set TS, as illustrated in Figure 2.5. The second feature is defined as $V(s) = \mathcal{T}(S(s))$ [23]. With these data the test configuration is complete and the production phase of the CDT can start. Algorithm 1 presents the ICI-based CDT pseudocode.

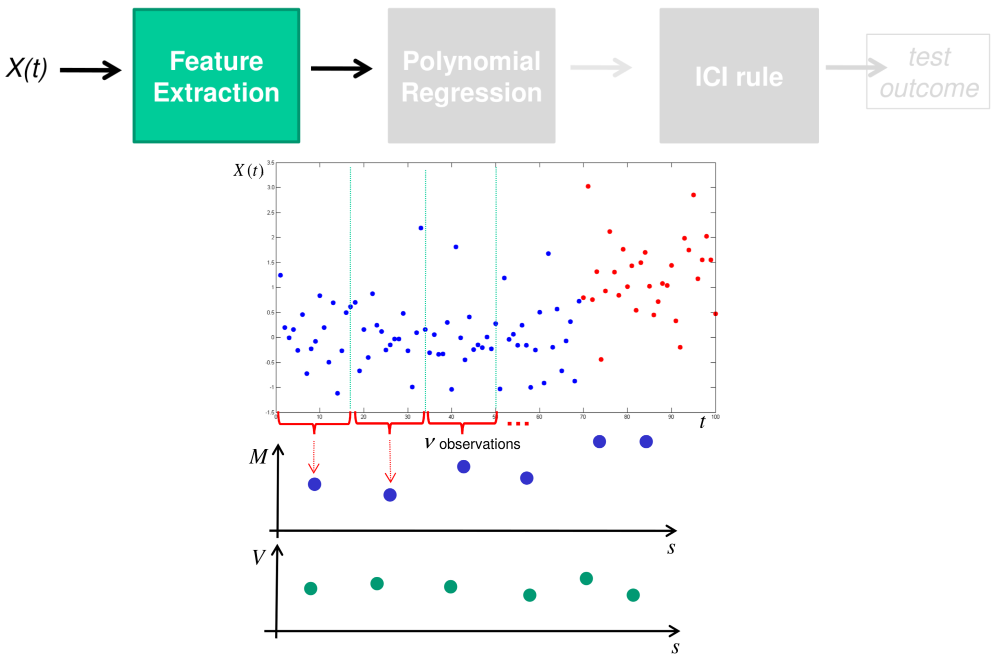
Algorithm 1 CDT using the ICI rule

```
    **TEST CONFIGURATION**: once all the observations belonging to the training phase have been collected
 1: Compute the sample mean for every observations window $s$ belonging to the training phase $TS$ using 2.6 as: $\{M(s), s = 1, \dots, S_0\}$, where $S_0$ is the number of training windows.
    For the feature **Sample Mean**:
 2:     Compute the mean using 2.3 as: $\hat{\mu}^M_{S_0} = \sum_{s=1}^{S_0} \frac{M(s)}{S_0}$;
 3:     Compute the variance using 2.3 as: $\hat{\sigma}^M_{S_0} = \sqrt{\sum_{s=1}^{S_0} \frac{(M(s)-\hat{\mu}^M_{S_0})^2}{S_0-1}}$;
 4:     Compute the Confidence Interval using 2.2 as: $\mathcal{I}^M_{S_0} = [\hat{\mu}^M_{S_0} - \hat{\sigma}^M_{S_0};\ \hat{\mu}^M_{S_0} + \hat{\sigma}^M_{S_0}]$;
    For the feature **Sample Variance**:
 5:     Compute the quadratic deviation for every window $s$ belonging to the training phase using 2.7 as: $\{S(s), s = 1, \dots, S_0\}$, where $S_0$ is the number of training windows;
 6:     Compute $h_0$ as defined in Algorithm 2;
 7:     Compute the power-law transform for every window $s$ belonging to the training phase using 2.8 as: $\{V(s) = \mathcal{T}(S(s)), s = 1, \dots, S_0\}$, where $S_0$ is the number of training windows;
 8:     Compute the mean using 2.3 as: $\hat{\mu}^V_{S_0} = \sum_{s=1}^{S_0} \frac{V(s)}{S_0}$;
 9:     Compute the variance using 2.3 as: $\hat{\sigma}^V_{S_0} = \sqrt{\sum_{s=1}^{S_0} \frac{(V(s)-\hat{\mu}^V_{S_0})^2}{S_0-1}}$;
10:     Compute the Confidence Interval using 2.2 as: $\mathcal{I}^V_{S_0} = [\hat{\mu}^V_{S_0} - \hat{\sigma}^V_{S_0};\ \hat{\mu}^V_{S_0} + \hat{\sigma}^V_{S_0}]$;
11: Set the current working windows number to the last training window: $s = S_0$;
    **PRODUCTION PHASE**: loop repeated for every new collected window
12: **while** ($\mathcal{I}^M_s \neq \emptyset$ && $\mathcal{I}^V_s \neq \emptyset$) **do** //Keep repeating as long as the confidence intervals is not empty;
    {
13:     Increase by one unit the current working windows number: $s = s + 1$;
14:     Wait for $\nu$ observations, until the current window $Y_s$ is populated;
15:     Compute the sample mean $M(s)$ for $Y_s$ using 2.6;
        For the feature **Sample Mean**:
16:         Compute the mean using 2.4 as: $\hat{\mu}^M_s = \frac{(s-1)\hat{\mu}^M_{s-1} + M(s)}{s}$;
17:         Compute the variance using 2.4 as: $\hat{\sigma}^M_s = \frac{\hat{\sigma}^M_{S_0}}{\sqrt{s}}$;
18:         Compute the Confidence Interval using 2.2 as: $\mathcal{I}^M_s = [\hat{\mu}^M_s - \hat{\sigma}^M_s;\ \hat{\mu}^M_s + \hat{\sigma}^M_s] \cap \mathcal{I}^M_{s-1}$;
        For the feature **Sample Variance**:
19:         Compute the quadratic deviation $S(s)$ for $Y_s$ using 2.7;
20:         Compute the power-law transform $V(s) = \mathcal{T}(S(s))$ for $Y_s$ using 2.8;
21:         Compute the mean using 2.4 as: $\hat{\mu}^V_s = \frac{(s-1)\hat{\mu}^V_{s-1} + V(s)}{s}$;
22:         Compute the variance using 2.4 as: $\hat{\sigma}^V_s = \frac{\hat{\sigma}^V_{S_0}}{\sqrt{s}}$;
23:         Compute the Confidence Interval using 2.2 as: $\mathcal{I}^V_s = [\hat{\mu}^V_s - \hat{\sigma}^V_s;\ \hat{\mu}^V_s + \hat{\sigma}^V_s] \cap \mathcal{I}^V_{s-1}$;
    }
24: Detect a change in $[(s-1)\nu, s\nu]$;
```

Algorithm 2 Computation of exponent h_0 for Gaussianizing Transform according to [62]

```
Compute the first six cumulants of $X$ from the Training Set $TS$
1:  Compute the first six raw Moments $m_k = \frac{1}{n}\sum_{i=1}^{n} X_i^k$, where $k = 1, \dots, 6$ and $n$ is the total number of samples belonging to $TS$;
2:  Compute the first six finite cumulants $C_n$ as an $n^{th}$-degree polynomial in the first $n$ raw moments $m_n$:
        $C_1 = m_1;$
        $C_2 = m_2 - {m_1}^2;$
        $C_3 = 2{m_1}^3 - 3m_1m_2 + m_3;$
        $C_4 = -6{m_1}^4 + 12{m_1}^2m_2 - 3{m_2}^2 - 4m_1m_3 + m_4;$
        $C_5 = 24{m_1}^5 - 60{m_1}^3m_2 + 20{m_1}^2m_3 - 10m_2m_3 + 5m_1(6{m_2}^2 - m_4) + m_5;$
        $C_6 = -120{m_1}^6 + 360{m_1}^4m_2 - 270{m_1}^2{m_2}^2 + 30{m_2}^3 - 120{m_1}^3m_3 + 120m_1m_2m_3 +$
            $- 10{m_3}^2 + 30{m_1}^2m_4 - 15m_2m_4 - 6m_1m_5m_6;$

Compute $h_0$ (as described in [62]):
3:  Compute the first three cumulants of $Y = S^2/\sigma^2$ [62] as:
        $\kappa_1 = n - 1;$
        $\kappa_2 = (n-1)^2[C_4/(n{C_2}^2) + 2/(n-1)];$
        $\kappa_3 = (n-1)^3[C_6/n^2 + 12C_4C_2/\{n(n-1)\} + 4(n-2){C_3}^2/\{n(n-1)^2\} +$
            $+ 8{C_2}^3/(n-1)^2]/{C_2}^3;$
    where $n$ is the total number of samples belonging to $TS$;
4:  The exponent $h_0$ is defined as:
        $h_0 = 1 - \frac{\kappa_1\kappa_3}{3{\kappa_2}^2}.$
```

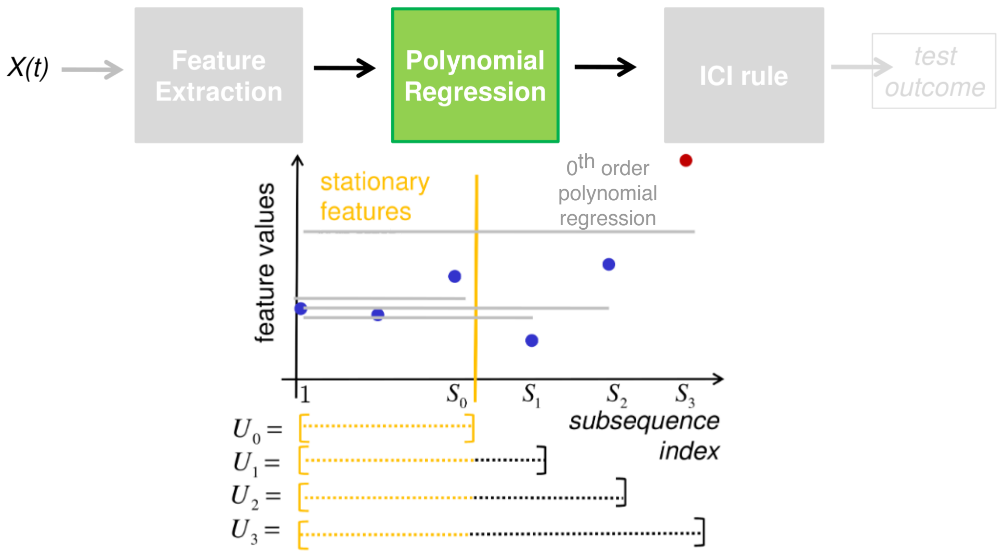


(a) *Upcoming observations are partitioned in non-overlapping subsequences of ν observations and features are computed on each subsequence. The small blue dots represent the observations, they turn red after the change. The bigger blue and green dots represent the extracted features, respectively for the sample mean and sample variance.*

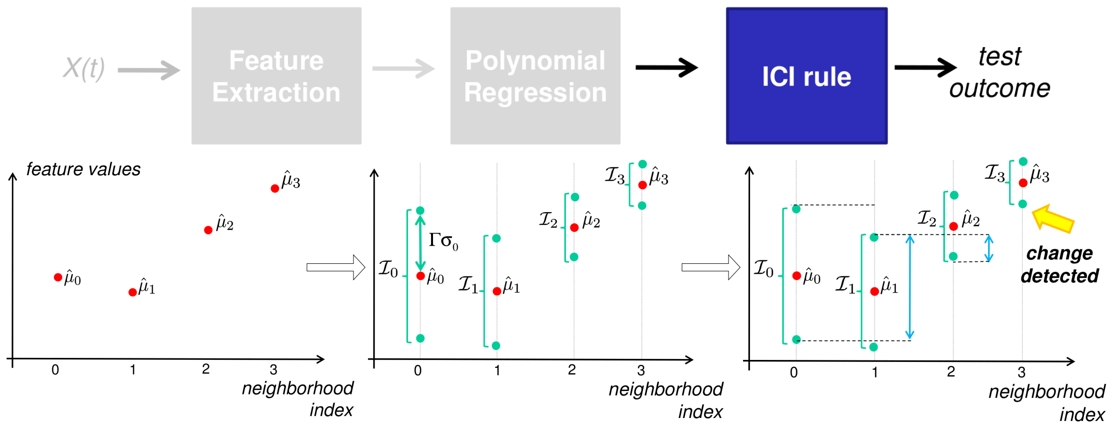
Figure 2.6: *First step of the ICI-based CDT. Picture taken from [41].*

2.4 Conclusions

In this chapter, we explained what tunable applications are, and we introduced background concepts on the subject of autotuning. We described the peculiarities of dynamic autotuners, and we analyzed the current state-of-the-art, to motivate the choice of mARGOt as target framework for our new module. In the second part of the chapter, we presented the broad topic of CDTs, illustrating some real-world use cases in which those techniques are employed. Finally, we introduced the ICI-based CDT, the algorithm underlying the first level test of our hierarchical CDT. In the next chapter, we will delve deeper into the functional details of the mARGOt framework.



(b) *Set of neighborhoods U_i having the leftmost extreme at $S = 1$. The estimates associated to each neighborhood are obtained by the 0^{th} order polynomial fit of feature values. The red dot belongs to the first subsequence where the change was detected.*



(c) *In the leftmost figure there are the feature estimates for each neighborhood. In the central picture the Confidence Interval (CI) is computed for every neighborhood. Here it is evident the interval-amplifying action of the parameter Γ. Finally, in the last figure the ICI rule selects the longest neighborhood for which the Intersection of Confidence Intervals is not empty and detects the change. The brackets represent the confidence intervals, while the arrows represent their intersection.*

Figure 2.6: *Last two steps of the ICI-based CDT. Picture taken from [41].*

CHAPTER 3

Background

This chapter describes background concepts and implementation details of the preexisting target autotuner that we will improve, which is the mARGOt framework, together with its module Agorà. The reasons behind such a choice have already been discussed in Chapter 2. Since the designated framework works on a distributed architecture, where the protagonists are local clients executing the application and a dedicated server in charge of managing the distributed execution, we approach the discussion going from the "local" components, towards the "remote" elements. The autotuner is the only component which can be used as standalone locally, while the Agorà module, as we will see, is an optional module, which benefits from the increased computational power offered by a parallel distributed architecture. For this reason, before switching to the remote part, we give an overview of the target distributed architecture, and on MQTT, the communication protocol behind the communications among the different components of the system.

3.1 mARGOt Autotuner - Local Component

The target application driven by the autotuner is tunable, meaning that it exposes at least one parameter which influences its behavior, and ideally it should repeatedly execute a task. A dynamic autotuner is an application-driver which, once in possession of the target application's knowledge, manages the application by setting its software-knobs to the configuration which best satisfies the run-time requirements. The dynamic autotuner, core module of the mARGOt framework, is grounded on the Monitor-Analyze-Plan-Execute (MAPE) feedback loop defined in [49]. As the name itself suggests, in this control loop we can identify four principal functions (Figure 3.1):

- Monitor: infrastructure which enables the application to self-adapt by sensing the

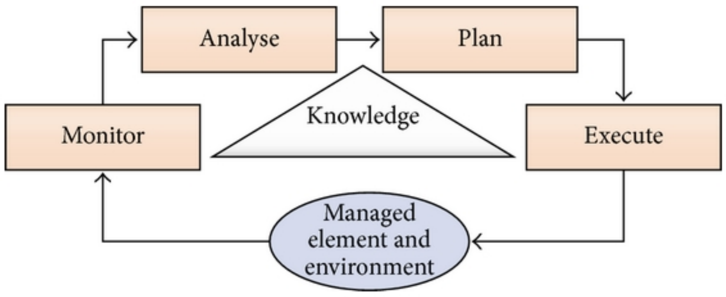


Figure 3.1: *MAPE control loop design*

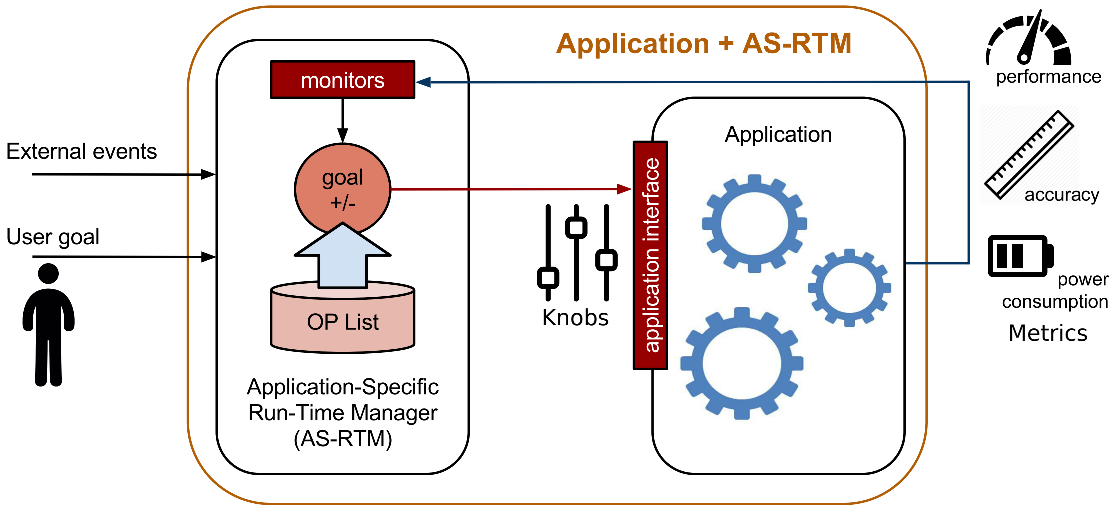


Figure 3.2: *mARGOt autotuner structure*

execution context and gathering runtime information on the observed application behavior, such as throughput, consumed power, execution time;

- Analyze: performs data analysis and reasoning comparing the information provided by the monitor(s) against the pre-defined goals;
- Plan: is the heart of the autotuner and it is represented by the Application-Specific RunTime Manager (AS-RTM) which is in charge of automatically tuning the application knobs according to its requirements and to the design-time and run-time knowledge;
- Execute: the application itself is in charge of executing the planned action, with the configuration enforced by the AS-RTM.

The task of driving the target application is fulfilled by the AS-RTM module which selects the best configuration, taking into account the information coming from the monitors. The knowledge on the possible working configurations is represented by a list of Operating Points (OPs), thus the AS-RTM selects the best OP in the list.

Each OP represents a single configuration for the application in terms of software knobs (list of parameters), enhanced with the performance profiled at design-time (metrics values). Thus the OPs are the result of a modeling phase which aims at profiling the application behavior by discovering rules that correlate the application knobs (rule head) with the application performance metrics (rule body); rules of the kind:

$$\text{configuration of tunable knobs} \xrightarrow{yield} \text{performance metrics}$$

for instance:

$$\text{Executing the application using 1 thread} \xrightarrow{yield} \text{Execution time: } 60\text{ms}$$

$$\text{Executing the application using 2 thread} \xrightarrow{yield} \text{Execution time: } 30\text{ms}$$

The list of OPs derives from a DSE phase; all the points are Pareto-optimal with respect to the target metrics. This is not a strict requirement, since the list of OPs represents all the possible available configurations which can be applied; the point is that the Pareto filtering obviously reduces the number of OPs and thus the overhead for selecting the best OP among all the available ones.

mARGOt has reaction policies according to changes in both application requirements and performance. The monitor infrastructure provides feedback information that enables the reaction to changes in the observed metrics at run-time. The monitors can be of various nature: time, throughput, memory, CPU usage, among others.

A concept linked to the monitors is the one of the Goal, which represents a single requirement of the application on a monitored metric, e.g. for a video processing application it might be "The average throughput must be greater than 3 frame per seconds". At runtime the application is able to check whether the goal is achieved.

As mentioned in Section 2.2, there are applications for which the metrics of interest are deeply input-data-dependant. If the inputs are not ordered, the situation can deteriorate dramatically, given that the reactive policies are not enough. The linear correction enforced by the run-time information provider will likely not be able to cope with a continuous variation of the input. These situations require particular care not to violate the assumption of independence and identical distribution (i.i.d.) of the observed data; this is a requirement for both the autotuner itself and for the CDT mechanism that we will introduce with this work. In order to restore the i.i.d. assumption, this question has to be taken care of at model-level. This translates in better exploiting the design-time knowledge related to the input features, e.g. input data size or input data autocorrelation. We need to take them into account in the decision making process to increase the proactivity in their respect, thus employing an OP structure of the kind:

$$\langle inputfeatures,\ knobs\, configuration\rangle \implies \langle metric1,\ metric2,\ ...,\ metricN\rangle$$

At the moment it is required for the user to manually specify the data-feature clusters, therefore for instance if the user knows beforehand that the number of rows that their application is going to process can be either 1000 or 3000 or 10000, then they need to explicitly indicate these three clusters. In such cases the framework uses a Data-Aware AS-RTM, which instantiates a different AS-RTM for each feature cluster; where the AS-RTM is the interface towards the application. This approach enables the proactive policy with respect to the input data features.

Figure 3.3 provides a symbolical representation of the different components which compose the mARGOt autotuner. It shows a AS-RTM for every feature cluster. Every AS-RTM is composed of the corresponding application knowledge, Runtime Information

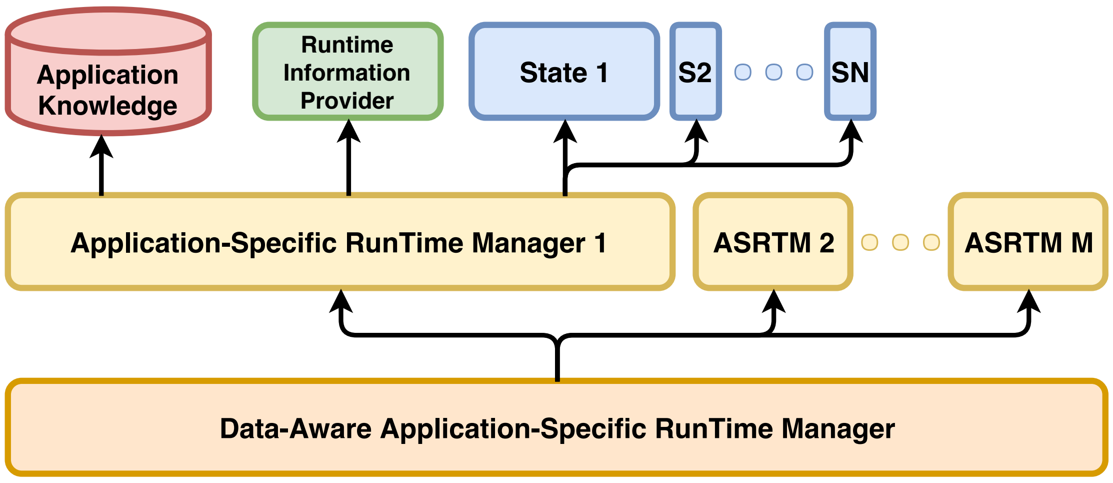

Application Knowledge
Runtime Information Provider
State 1
S2
SN
Application-Specific RunTime Manager 1
ASRTM 2
ASRTM M
Data-Aware Application-Specific RunTime Manager

Figure 3.3: *mARGOt Data-Aware AS-RTM with M feature clusters.*

Provider, and states at which the application can be configured. The Runtime Information Provider is connected to the monitoring infrastructure and allows the actuation of reactive policies by means of linear corrections with respect to the current application knowledge. The application knowledge represents the model of the application behavior; it is assumed as ground truth and used by the autotuner. The main source of concern, that we want to address with this thesis, is that there exist situations which invalidate such knowledge. This is the reason why we want to constantly assess it and, if needed, update it.

3.1.1 mARGOt Framework Toolchain

mARGOt is a lightweight C++ library which can be integrated with tunable C++ applications. The autotuner is the only standalone component of the whole framework. It is integrated directly with the application, on the local client. We encourage to read the available framework documentation [11] to better understand the integration process between the application and the mARGOt framework.

The separation of functional and extra-functional concerns has been adopted; the former are the aspects related to the specific target application, the algorithms and the kernels themselves, while the latter are the features associated with the framework integration, as shown in Figure 3.4.

In order to facilitate the integration of the autotuner with the application, the framework provides a tool that automatically generates the required code from XML configuration files. In particular, the developer has to necessarily provide the configuration file where they express the desired monitors which provide runtime information over the metrics, the metrics of interest themselves, the goals and constraints to allow the autotuning. The configuration files are then parsed and interpreted through a Python wrapper which outputs the necessary glue code that allows the integration between the application and the autotuner, as shown in Figure 3.4. mARGOt also provides a high-level interface tailored for the target application.

Moreover, as clarified before, the main requirement for the autotuner to work is to have knowledge of the possible working configurations of the target application, in

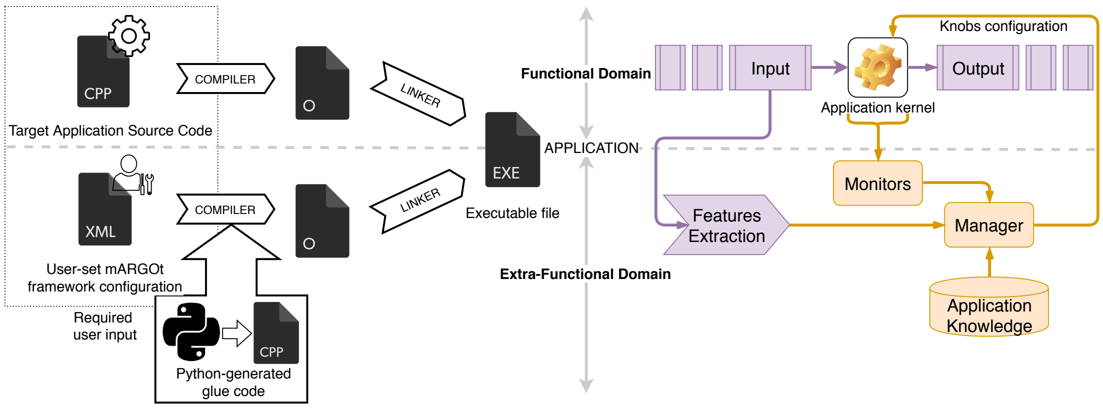


Figure 3.4: *Outline of the integration process between the target application and the mARGOt framework. The user has to provides the (specifically edited) source code of their application and the framework XML configuration file. Highlighted is the separation between functional and extra-functional domains.*

the form of OPs. If the autotuner is used as a standalone module, then the user has to “manually” provide the list of OPs, pre-computed externally. The computation of the OPs is complex.

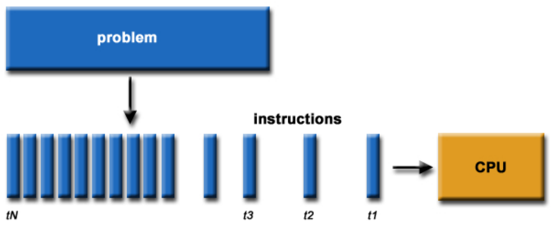
Here comes into play the server-side Agorà module, provided with the mARGOt framework, which is a component that relieves the final user from the burden of computing and providing the OPs. It learns the application knowledge online in a distributed fashion. When relying on the Agorà module to compute the application model, the user also needs to state the parameters of Agorà in the XML configuration file, e.g. the DoE technique, the knobs to be explored and their ranges, the metrics to be predicted, the input features which need to be taken into account for the final model. But this will be clearer after having fully explained the working mechanism behind Agorà.

Let us specify that our contribution targets the use case in which Agorà is used to compute the application model. Our approach re-uses mARGOt’s infrastructure to adapt the application knowledge, by commanding a re-learning of such knowledge when needed.

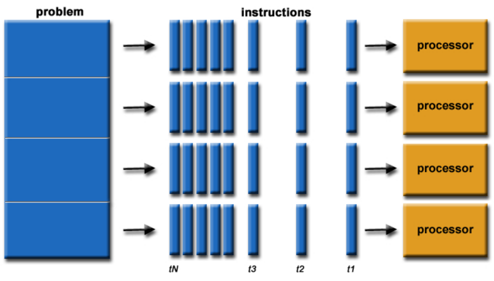
3.2 Target Architecture

mARGOt was originally designed for the autotuning of embedded platforms. Naturally, it can be also be applied to High Performance Computing (HPC) systems. Actually, the HPC scenario is the one on which we focus since it benefits the most from the distributed nature of mARGOt. It is relevant to point out that, even though our target is mainly the HPC architecture, this is because it is the most sensible and common use case in which autotuner frameworks are employed, and specifically referring to our case, we will see how Agorà is able to take advantage of having multiple distributed nodes on which to carry out the DSE phase. That said, in principle the overall mARGOt framework we propose can be deployed on any architecture.

High Performance Computing refers to the practice of aggregating computing power in a way that delivers much higher performance, in order to solve large problems in science, engineering, or business [18]; this is achieved through the use of parallel



(a) *Serial computing*



(b) *Parallel computing*

Figure 3.5: *Serial and parallel computation of a problem.*

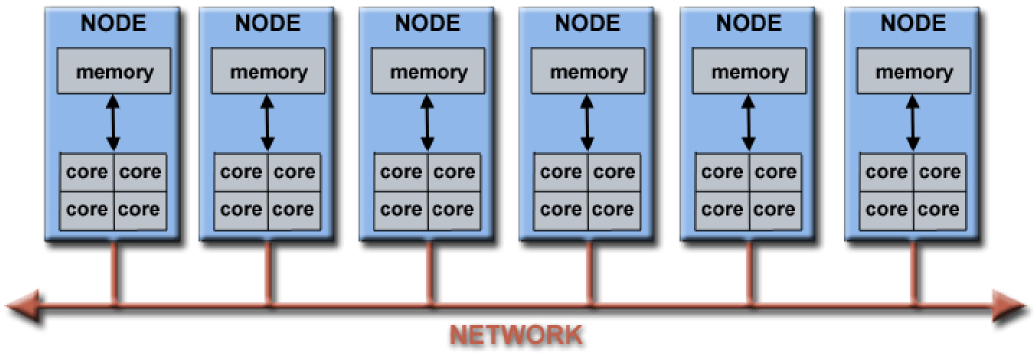


Figure 3.6: *Parallel architecture schema.*

processing. The term HPC applies especially to systems that function above a teraflop, or 10^{12} Floating-Point Operations Per Second (FLOPS) [10] and is occasionally used as a synonym for supercomputing, although technically a supercomputer is a system that performs at, or near, the currently highest operational rate for computers (as of November 2018 “Summit” is the fastest supercomputer in the world, capable of 200 petaflops, or 200×10^{15} FLOPS [16]). High-performance systems often use custom-made components in addition to so-called commodity components (which involve the use of large numbers of already-available computing components for parallel computing, to get the greatest amount of useful computation at low cost).

HPC applications run on massively parallel systems and consist of concurrent programs designed using multi-threaded, multi-process models. Parallel computing (as opposed to the traditional serial computing) simultaneously uses multiple resources in order to solve a computational problem, taking advantage of concurrency, that is the possibility to execute, at the same time, those parts that are independent of each other. In this way large problems can (often) be divided into smaller ones, which can then be solved concurrently (Figure 3.5). A typical parallel architecture is composed of an arbitrary number of computers (nodes), each of them with multiple processors, cores, functional units, connected by a high-bandwidth low-latency network (Figure 3.6) to form a larger parallel computing cluster [30].

Generally, supercomputer system design can be divided in two categories according to the computing model employed in the single nodes [40]: homogeneous and heterogeneous solutions. Heterogeneous computing refers to systems that use more than one kind of processor. These systems gain performance or energy efficiency not just by adding the same type of processors (homogeneous), but by adding dissimilar co-processors, usually incorporating specialized processing capabilities to handle particular tasks. As both Central Processing Units (CPUs) and Graphics Processing Units (GPUs) become employed in a wide range of applications, it has been acknowledged that both of these Processing Units (PUs) have their unique features and strengths and hence, CPU-GPU collaboration is inevitable to achieve high-performance computing. This has motivated a significant amount of research on heterogeneous computing techniques, along with the design of CPU-GPU fused chips [60]. This led to the emergence of General-Purpose computing on Graphics Processing Units (GPGPU), which is the use of a GPU to perform computation in applications traditionally handled by the CPU.

Working with heterogeneous architectures presents new challenges not found in typical homogeneous systems [50]. Nevertheless, the gain in performance and efficiency of such systems is clearly proved by their ever-increasing adoption in modern computing systems. As demand for processing power grows, HPC will likely interest businesses of all sizes, particularly for transaction processing and data warehouses, in addition to all the many multidisciplinary areas in which is already largely employed including biosciences, geographical data, oil and gas industry modeling, climate modeling and media and entertainment. We will not delve deeper into the details on how parallel applications or parallel architectures work; they pose considerable methodological challenges, since they bring together several technologies such as computer architecture, algorithms, programs and electronics, thus the topic is very broad and it is out of the scope of this work.

3.3 MQTT Messaging Protocol

This section provides a short overview on the Message Queuing Telemetry Transport (MQTT) messaging protocol, since it underlies all the communications among the different distributed components in our framework. MQTT is a publish-subscribe-based protocol. It is designed for connections with remote locations where a “small code footprint” is required or the network bandwidth is limited and for this reason it is widely used in Internet of Things (IoT) and Machine to Machine (M2M) applications [14].

An MQTT system consists of clients communicating with an intermediary server, known as “broker” (Figure 3.7). In software architecture, *publish-subscribe* is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes, known as *topics*, without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are. When a publisher has a new item of data to distribute on a given topic, it sends a control message with the data to the broker. The broker then distributes the information to any client that has subscribed to that topic; in other words, the broker performs the filtering using the topic as routing information. It implements a store and forward function to route messages from publishers to subscribers.

This pattern provides great network scalability and is suited for networks whose topology changes dynamically. A topic is a logical channel, implemented as a string that can have more hierarchy levels, each of these separated by a forward slash “/”

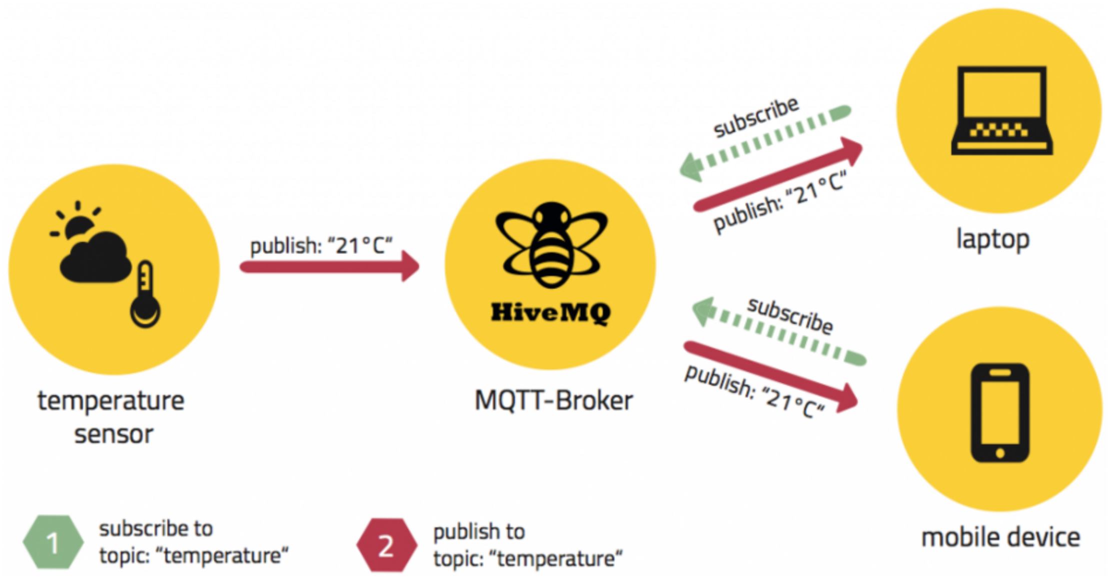


Figure 3.7: *MQTT publish/subscribe example [12].*

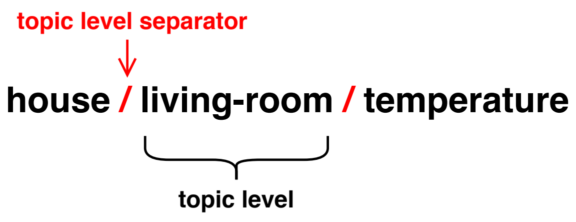


Figure 3.8: *MQTT topic hierarchy example.*

(Figure 3.8). Wildcards can be used to subscribe to multiple topics at once: the single-level one is denoted with the symbol "+", while the multi-level one with the symbol "#". These allow arbitrary topics in their place, with the only difference that the single-level obviously enables any topic just for the level in which it is currently inserted, whereas the multi-level one enables the entire sub-tree of all the possible topics for its level and the following sub-levels, as shown in Figure 3.9.

A very interesting feature offered by the MQTT protocol is the Last Will and Testament (LWT), that is the ability to notify other clients about an ungracefully disconnected client [13]. When a publishing client first connects to the broker, it can set up a default message to be sent to the subscribers if the broker detects that the publishing client has unexpectedly disconnected from the broker. We will see how we used this feature in our framework to detect crashes and disconnections of different modules and clients and to react accordingly.

3.4 Agorà - Remote Module

The autotuner works by selecting the best configuration for the application, and for this to happen it has to have an application-specific knowledge in the form of OPs. It

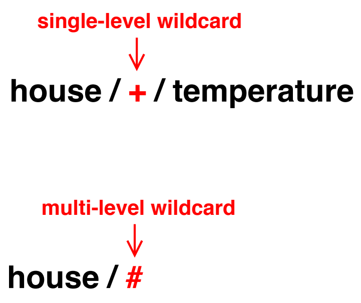


Figure 3.9: *MQTT single and multi-level topic wildcards example.*

is agnostic with respect to how such OPs are computed. This implies that, normally, there has to be a preceding offline phase to profile the application behavior. The Agorà module [54] aims at offloading the user of the burden of computing the OPs to be provided to the autotuner by building and managing the application knowledge online, during the execution itself. This is achieved by performing the application Design Space Exploration (DSE) at run-time in an automated and possibly distributed fashion, exploiting multiple clients running the same application, if present, thus considerably speeding up the overall workflow. The idea behind the modus operandi of Agorà is to drive the application to execute a subset of parameters configurations taken from their design space, result of a Design of Experiments (DoE) study, in order to gather all the values of the metrics of interest associated to them. This list of configurations composes the training set that Agorà is going to use for the prediction of the complete application model -namely the list of OPs to be fed to the autotuner- which is later derived thanks to Machine Learning techniques.

3.4.1 Agorà Module Structure

While the autotuner is a module which is integrated with the target application locally on the client where the application itself lies, Agorà has been designed to reside on a central server, and it interacts with the autotuner using a client-server model. There are multiple reasons behind this design choice, of these the key ones are:

- The machines running the target applications are not slowed down by the additional computation overhead introduced by Agorà for the model computation;
- Agorà can exploit simultaneously all the clients running the same application to perform the DSE, thus substantially reducing the time required to produce the model of the application behavior.

Figure 3.10 illustrates the interaction between the server and the clients in a distributed environment. On the server there is a “Remote Application Handler” for every managed application, since Agorà is able to supervise multiple different applications at the same time, while on every client there is a “Local Application Handler”.

Figure 3.11 shows a simplified version of the client structure. We can notice that the autotuner manages the application according to its own “knowledge”. The “Local

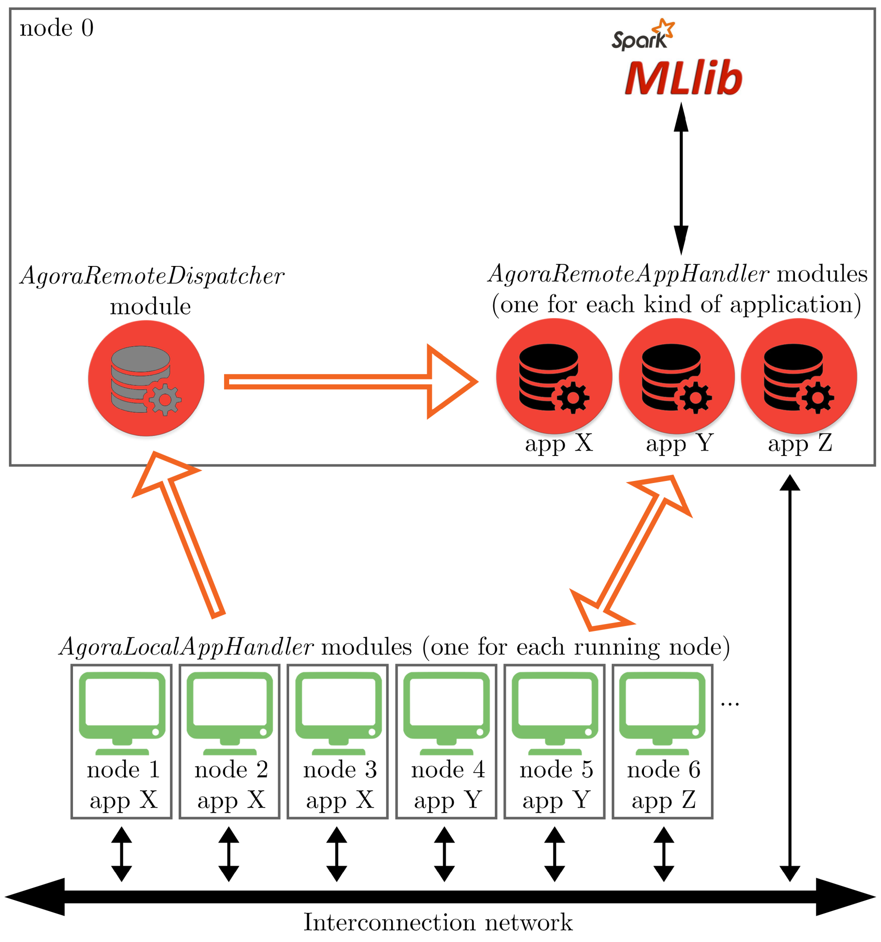

node 0
Spark MLlib
AgoraRemoteDispatcher module
AgoraRemoteAppHandler modules (one for each kind of application)
app X
app Y
app Z
AgoraLocalAppHandler modules (one for each running node)
node 1 app X
node 2 app X
node 3 app X
node 4 app Y
node 5 app Y
node 6 app Z
...
Interconnection network

Figure 3.10: *Overview of Agorà in a distributed environment [54].*

Application Handler" is in charge of communicating with the server, i.e. receiving the configurations to explore or the final model, sending back the observed metrics, and setting the local client knowledge. The user has to provide Agorà with configuration parameters, such as the DSE technique of choice and the range of available values for every exposed knob of the target application. These are specified in the XML configuration file mentioned in Subsection 3.1.1.

Agorà's workflow when a new application has to be profiled, presented in Figure 3.12 and Figure 3.13, is the following:

1. According to the user-specifications Agorà computes the DoE. Then the online application DSE begins;

2. The configurations belonging to the DoE are sent by the server (Agorà) to the clients running the application. Every client is forced to run the received configuration. The autotuner works locally on each client with its own knowledge that in standard use should theoretically contain the OPs. Hence we set the local client knowledge to just the assigned configuration to explore;

3. Agorà later collects all the observations coming from the clients recording them as

$$\langle knobs\ configuration \implies observed\ metrics \rangle$$

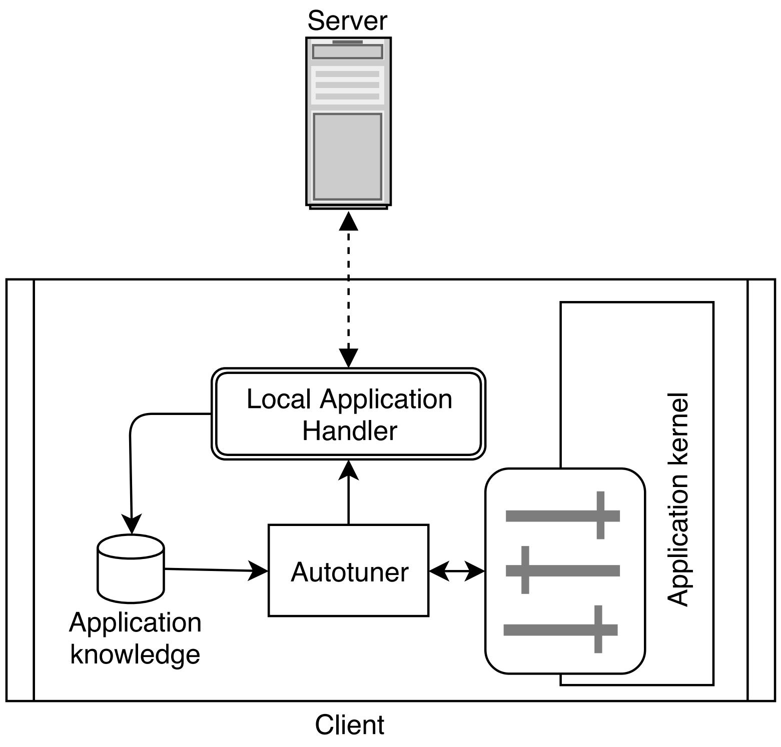


Figure 3.11: *Overview of the local client structure.*

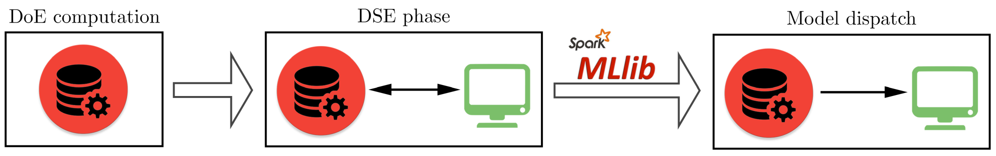


Figure 3.12: *Overview of Agorà workflow when a new application is profiled [54].*

and this goes on until all the configurations belonging to the DoE have been explored (go back to (2));

4. When the DSE finishes Agorà computes the application model through machine learning techniques. Once ahold of the model the server distributes it to all the clients.

All the communications between the server and the clients use the MQTT protocol (Section 3.3). The application name is used as a MQTT topic level to allow the correct routing of messages between the corresponding local and remote handlers. Eclipse Paho [7] is the open-source implementation used for the C++ client libraries, while Eclipse Mosquitto [6] is open-source implementation used for the broker server.

Figure 3.14 presents Agorà's server-side components. Cassandra [2] is the open-source NoSQL DataBase Management System (DBMS) designated to deal with the persistence of the generated knowledge; meaning that once an application model is computed it is also stored. In this way it can be reloaded and sent to potential new nodes which start executing the corresponding application, without the need of repeating the DoE and DSE phases. It is worth mentioning that among others, there is a table which

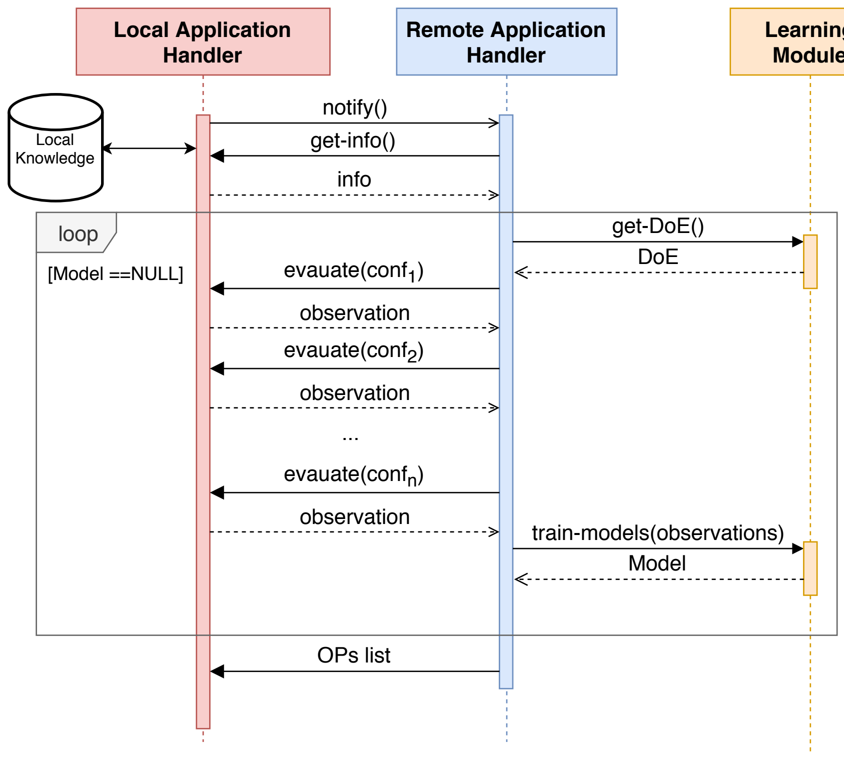


Figure 3.13: *Sequence diagram showing Agorà's server-side and client-side workflow when profiling an application.*

contains the original DoE, and a "trace" table, which keeps track of all the observations coming from the applications both during the online training and the production phases. These implementation details are needed to better understand the realization of our methodology in Chapter 5.

Agorà has been designed in such a way that it remains orthogonal with respect to the machine learning technique used to extract the model from the collected observations after the DSE.

3.5 Conclusions

In this chapter, we illustrated the operating principle of the two main components that constitute the mARGOt framework: the local autotuner, coupled with the target application on the clients, and the remote module Agorà, in charge of the online learning of the application knowledge. We presented the mechanisms behind the reactive and proactive policies used by mARGOt and the process of integration of the target application with the framework. We also provided an overview on the HPC architecture that, being composed of many nodes, represents the ideal system able to fully exploit both the strategy employed by Agorà and the one we will propose in this work for the continuous monitoring of the execution of the autotuned applications. We also explained the functioning of the MQTT protocol, which is at the base of the communications between the distributed components of the system. In the next chapter, we will introduce the methodology and the design choices underlying the working principles of the new

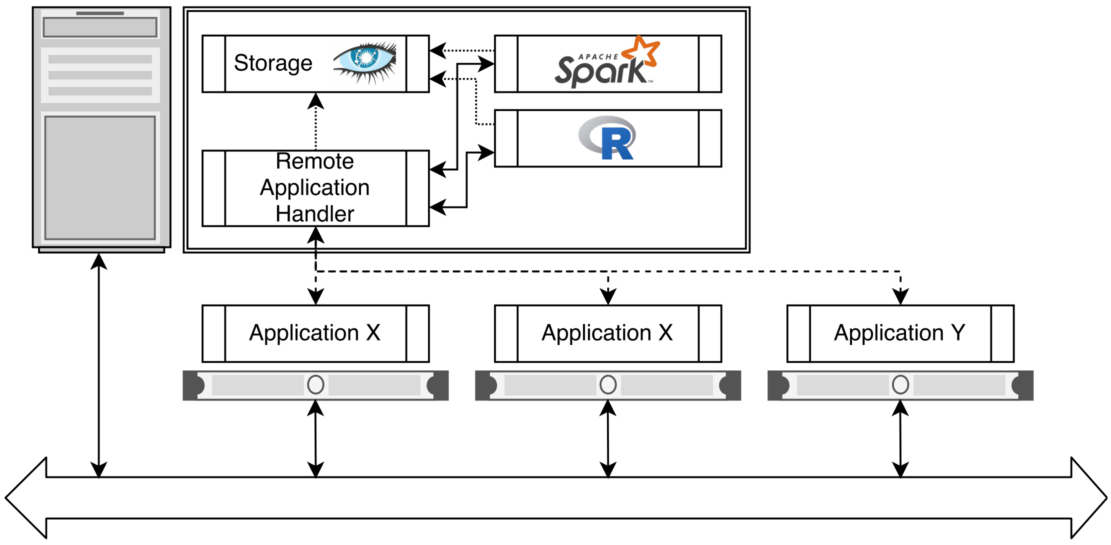


Figure 3.14: *Overview of Agorà's server structure.*

remote module.

CHAPTER 4

Methodology

We want to address the challenge of updating the application knowledge used by an autotuner framework to drive the application, when this knowledge proves to not be representative and sound with respect to the current context. This is necessary otherwise very likely efficiency, or, in the worst cases, the correctness of the computation itself, will be negatively affected. The first contribution of this work was in the creation of a new module for an autotuner framework which implements a technique to understand when this knowledge refresh is needed. The second contribution was the actual integration of this new module on top of a preexisting framework, together with all the mechanisms to execute the knowledge update ordered by our module.

This resulted in the “Beholder”[1] module, whose main goal is to detect, by means of a two-level hierarchical Change Detection Test (Section 2.3), whether any change in the observed application behavior, with respect to the estimates retrieved from the model, has happened. Assumption of such methodology is that the model in use represents the application behavior in its “stationary” state. We engineered this new module to be integrated on top of the mARGOt framework. The reasons behind this choice have been discussed in Chapter 2, and both mARGOt and its module Agorà have been described in Chapter 3.

Figure 4.1 shows the proposed enhanced mARGOt framework. The strategy of the Beholder is to evaluate incoming data from the autotuned nodes running the application. The robustness of the current model is evaluated enforcing policies which take into account the collective behavior of the nodes. With the Beholder we achieve awareness of a change at a central level, this enables the implementation of policies which weighting the percentage of “bad” (in which a change has been detected) versus “good” (in which a

[1]The meaning of the word “Beholder” is spectator, onlooker; it gains awareness of things by watching and listening. The Beholder is also a fictional monster in the Dungeons & Dragons fantasy role-playing game. Its appearance is that of a floating orb with a large mouth, a single central eye, and many smaller eyestalks on top with powerful magical abilities.

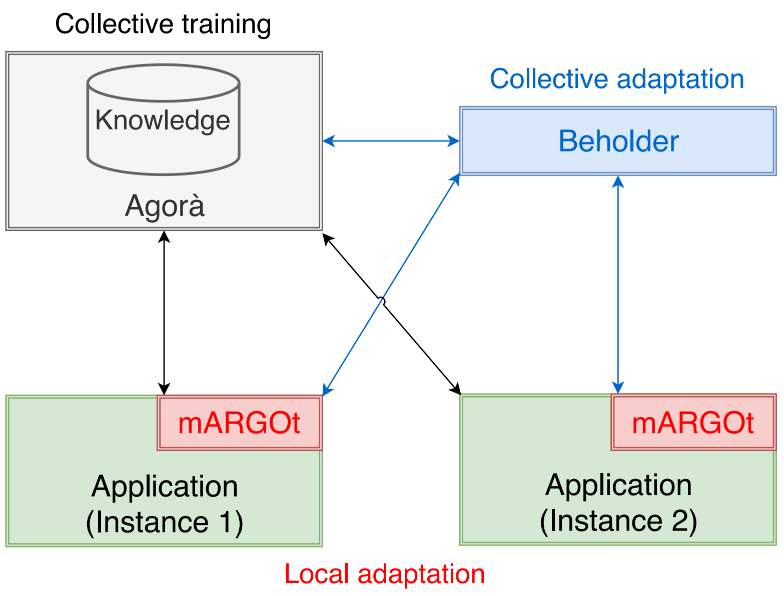


Figure 4.1: *Overview of the modules of the proposed enhanced mARGOt framework. The collective training represented by Agorà and the local adaptation performed by mARGOt constitute the preexisting baseline; the collective adaptation assigned to the Beholder module is the proposed novelty.*

change has not been detected) nodes. According to such percentage and to the end-user requirements, the Beholder can trigger a knowledge update exploiting the Agorà module; this required some modifications to the latter, to make it able to interface with the Beholder.

The advantages that we have from being aware that something has changed are several. The framework gains in proactivity, for instance, new nodes which start running an application receive always an up-to-date model as a starting point on which to work, and do not risk being given a list of OPs, now invalidated, which belonged to a different execution context. The autotuner, which lies locally on the client, is already reactive, meaning that it keeps correcting its own knowledge in order to cope with possible variations of the target application behavior, thanks to the feedback information provided by the MAPE loop (Section 3.1), but these adjustments are performed only locally by the clients themselves and are not shared on a global scale. Besides, the autotuner only tries to cope with the variation, through a linear scaling of the model prediction, proportional to the perceived deviation. Then, when anomalies are not linear, and thus not easily "predictable", the correction is very often wrong anyway, aside from the fact that it takes time to possibly converge to the correction.

This chapter presents the methodological choices with which we designed, from scratch, the Beholder module, bearing in mind that it had to be integrated into a preexisting framework. The implementations details will be described in Chapter 5. Before diving right into the core of the Beholder, we introduce other contributions to the framework, needed to allow a better integration of the whole workflow and to expose new functionalities to the final user.

```xml
<agora ..... >
    <pause polling_time="20" timeout="10000"/>
    .....
</agora>
```

Figure 4.2: *Snippet of an XML configuration file showing the definition of the parameters which control the busy-waiting mechanism to force the initial synchronization of the target application with Agorà. Both the parameters are expressed in milliseconds.*

4.1 Overview of the interaction with the autotuner framework

As a consequence of the integration of the new module, we had to introduce additional features in the existing framework, regarding the learning of the application knowledge. The first change is related to the input set management. We saw how Agorà takes care of the training in a fully automatic way. Nevertheless, there could be cases in which the user needs to differentiate the datasets used for training and production. So far this is not possible, then we will introduce such option.

Another issue is raised by those metrics whose computation is very intensive, thus not possible during the production phase, e.g. the evaluation of the error introduced by the approximate computation in use. Therefore, a further missing feature that prevents the achievement of a totally framework-controlled seamless execution, with potential multiple retraining phases during the whole application lifetime, is the lack of a mechanism which automatically controls such metrics, enabling their computation just during the (re)training phases.

4.1.1 Sync between target application and Agorà module

When the user relies on Agorà to compute the application knowledge needed to carry out the autotuning, the framework performs the necessary DSE at run-time, to gather enough information to build the model for the target application behavior and provide the nodes with the OPs. Typically, when the application is launched, the communication between the local node and the server, where respectively the application and Agorà lie, takes time to be set up over a network. Thus, we introduced the possibility for the user to set the application to wait for instructions from Agorà, before starting an uncontrolled computation. Once the synchronization between the local node and the server is complete the waiting is over. We expose two parameters which control this "busy-waiting": the frequency between the checks for Agorà's availability and an overall timeout before giving up the waiting and starting the computation anyway; both of these are expressed in milliseconds. The user is able to setup such parameters in the "pause" element of the XML configuration file (Subsection 3.1.1), under the "agora" parent element, as shown in Figure 4.2.

Let us remember that this synchronization between the local application and the server is only carried out once when the application starts, thus the overhead introduced with such waiting mechanism should be negligible, taking into account that HPC applications run typically for long periods of time.

4.1.2 Runtime Differentiation between Training and Production phases

The autotuner relies on a model of the target application to drive the latter, and when modeling techniques are involved, typically the best practice requires to perform the training with a set of representative inputs which covers pretty much all the future possible cases. In this way, the model is able to retrieve an accurate prediction in most cases. One of the benefits of relying on Agorà is in its ability to use a part of the production phase data as training to relieve the user from providing a dataset tailored for training. Anyway, the user may be willing to explicitly provide a specific, and more representative, training dataset; or they may be interested in changing their application workflow in a custom way according to the context. In order to allow for such customization, we implemented a straightforward boolean method exposed to the user, "`has_model()`", thanks to which they can inquire, from the application, the current status of the framework knowledge and behave accordingly.

4.1.3 Error Monitor

mARGOt already supports custom monitors that the user can use to record parameters gathered manually, possibly because application, or domain, specific. In this way, there is the option to have feedback information about, for instance, the accuracy of the computation, or, in other words, the error. There are metrics whose monitoring comes almost for free, for example, the execution time or the CPU/memory utilization monitors introduce no appreciable overhead, and thus can be used at runtime for an accurate check on actual real data. On the contrary other metrics come at a very high cost. One of the most emblematic and useful evidence of such a case is the monitoring of the error. Since one of the many promising approaches employed by autotuners is based upon approximate computing (Subsection 2.1.1.2), often we want to satisfy a minimum level of user-set accuracy. However, monitoring the error introduced by using an approximation is typically very computation-intensive, since it requires the dual computation of the approximated version and the reference optimal version and a comparison of these two. Obviously, except for rare cases, such an approach is impracticable, since at this point it is more convenient to just carry out the single not approximated computation.

Thus, we need a model able to predict the error corresponding to a given configuration. This is an evidence in which the user must be given a way to differentiate, in the application source code, whether the framework is performing the training phase, or if the model is already built and then it is in the production phase. This is because during the training phase the error is always computed, so the user has to make the application carry out the dual (approximated and optimal reference) computation and provide the resulting accuracy measure. On the contrary, in the production phase, ideally, we want to save as much time and resources as possible, thus the error computation is entirely avoided and just the approximated version of the kernel is run. We rely, for the accuracy, on the estimate retrieved from the model. This results in a huge gain in efficiency, still allowing for a partial control over the accuracy. One may argue that during the training two computations are still needed, which, again, is worse than just the single optimal version in terms of efficiency. But this is counterbalanced by the fact that, typically, the training is done just once and it is a lot faster with respect to the real future production

```xml
<monitor name="my_error_monitor" type="Error">
  <enable frequency = "periodic" period = "10" />
    .....
</monitor>
```

Figure 4.3: *Snippet of an XML configuration file showing the partial definition of the Error Monitor.*

phase, which, for HPC applications, is usually very prolonged.

The user can theoretically use the above-mentioned "`has_model()`" function to differentiate the different training/production phase workflow, respectively with and without the error computation, but we also decided to provide an "Error" type of monitor, which takes better care of all what is related to the presence and absence of such information. A very trivial issue is the logging in the storage, inside the so-called "trace" tables: sometimes the value for the error is available and other times it is not. The advantages of this new monitor can be summarized as follows:

- The error computation is automatically controlled by the framework on the basis of a user-set parameter in the XML configuration file, which offers an "enable" element, whose attribute "frequency" can be either `ALWAYS` (error computation always enabled), `TRAINING` (error computation enabled only during the training phase) and `PERIODIC` (which is like training, but it also sometimes enables the computation during production phase with a frequency controlled by the attribute "period", which represents the number of cycles between two activations of the error computation). Figure 4.3 shows an excerpt from the configuration of an error monitor, in which the definition of the auto-enabler mechanism is straightforward.

- We expose a boolean method "`compute_error()`" automatically controlled by the framework according to the user setting. It can be directly used in the application source code to check if the application needs to compute the error. According to it the user is required to provide the error value.

- The framework automatically takes care of the Cassandra Query Language (CQL) insert query in the "trace" table according to the presence or not of the error attribute.

The contribution which the Beholder module brings to the framework enables to possibly discover anomalies in the accuracy metric, which is otherwise often never actually checked at runtime; this is because entirely relying on the model estimate to push the computation efficiency at the extreme.

4.2 Overview of the Hierarchical CDT Structure

Intelligent systems meant to operate in nonstationary environments have to detect possible changes to proactively interact with the environment and adapt to evolving non-stationary conditions. One of the most appealing solutions consists in monitoring the statistical behavior of data by means of nonparametric sequential CDTs (Section 2.3). These tests assess the stationarity of the data-generating process without requiring any a priori information about the process or the change. One of the main drawbacks of

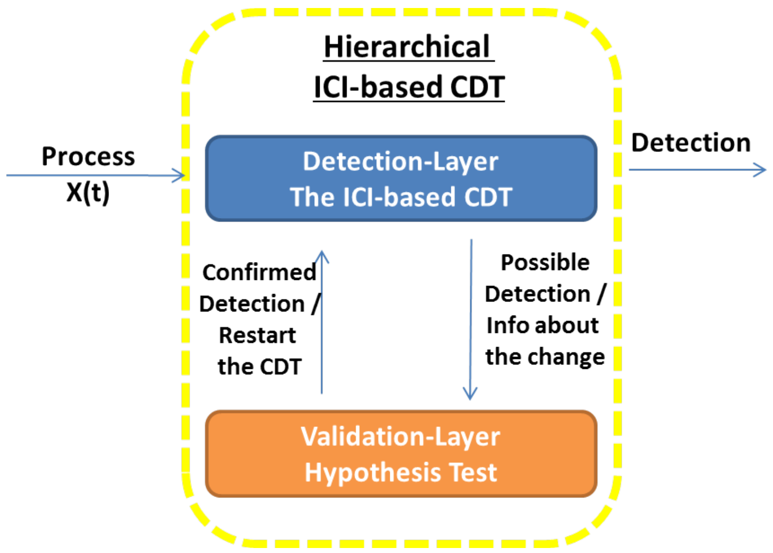


Figure 4.4: *Illustration of a hierarchical (two-level) CDT. Picture taken from [24].*

CDTs are false positives, i.e., detections not corresponding to an actual change in the data-generating process, as these induce intelligent systems to raise false alarms [24].

We propose an hierarchical CDT which combines different techniques to detect and validate changes; it features a two-layered architecture consisting in a detection layer and a validation layer (Figure 4.4). The Detection Layer is designed to steadily analyze the data stream at a low computational cost, by means of an online and sequential CDT. The Validation Layer performs an offline analysis based on a hypothesis test to determine whether the detection corresponds to an actual change in the data-generating process or not (False Positive detection). From now on we will use interchangeably the names “first level test”, “detection layer” and “ICI CDT”; the same goes for “second level test”, “validation layer” and “hypothesis test”.

Please note that, a strong assumption for the correct functioning of the proposed test is that the monitored data can be described as a random variable independent and identically distributed.

4.3 First Level Test - Detection Layer

The Detection Layer operates online and performs a sequential analysis of the observations stream. Data are evaluated as they are collected, and further collection is stopped as soon as an anomaly is detected. The purpose of this layer is to provide a prompt detection of possible nonstationarities. It is implemented through and ICI-based CDT (Subsection 2.3.5).

Data collection Data are gathered from all the nodes running the same application without distinction; we decided to use this policy to validate the model against all the clients that are simultaneously using it in a unique test. This allows to automatically concentrate just on changes that are perceived on a more general scale with respect to an alternative client-wise anomaly detection, and, most importantly, enables for more frequent model-checking, thanks to the fact that there are multiple sources of

```
<agora ..... >
    <beholder metrics="exec_time,error_metric"/>
    .....
</agora>
```

Figure 4.5: *Snippet of an XML configuration file showing the definition of the metrics on which the Beholder module has to perform its analysis.*

observations that participate in the same data-generating monitored process, thus the buffers that we will present below are filled-in faster. We offer the user the option to state the metrics on which the beholder has to carry out its analysis, in case there are metrics to be left unmonitored. This option is implemented as a parameter in the XML configuration file (Subsection 3.1.1), more precisely the "beholder" element, under the "agora" parent element, as shown in Figure 4.5.

Residuals computation The Beholder computes for all the application (beholder-enabled) metrics the so-called *residuals*, which are the absolute difference between the model estimate for the metrics of interest and their actual observed values from the nodes. We use the absolute difference since, at this stage, we are not interested in understanding the "direction" of the change. Possible future expansions of our work could theoretically use the added information of the direction to enforce different behaviors.

Controller buffers The Beholder module uses as many server-side controllers ("Remote Application Handler") as the number of applications it manages, one for each application. In this controller, there are as many buffers as the number of metrics whose behavior has to be monitored. Thus every metric has a buffer of ν observations which, once complete, forms a subsequence on which the ICI-based CDT operates; optionally the user can set this "window-size" parameter.

Test workflow structure Algorithm 3 outlines the logic underlying the first level test for every single application (inside every Beholder Remote Application Handler), and it is the pseudocode counterpart of Figure 4.7. Each time the beholder receives an observation from a client, it triggers a potential test evaluation. The test is actually only carried out when at least one buffer is complete (lines 7 and 8 in Algorithm 3); until then it is just a matter of parsing the messages from the clients, computing the residuals and filling the corresponding buffers, as outlined in Figure 4.6.

Test outcomes When the first level test is performed, its possible outcomes are, either to proceed with the second level if a change has been detected (line 15 in Algorithm 3), or to empty the buffers and to return from the method (line 17 in Algorithm 3), to allow for future data to be gathered and form the next subsequence which is going to be tested. As explained in Subsection 2.3.5, a test monitors just one metric at the time. This entails that there are multiple tests which work in parallel when multiple metrics are being monitored. As soon as one of these detects an anomaly the second level is triggered, without waiting for a confirmation of the detected change from other metrics, if present (Figure 4.7). We decided to rely on the second level test for a possible discard of the

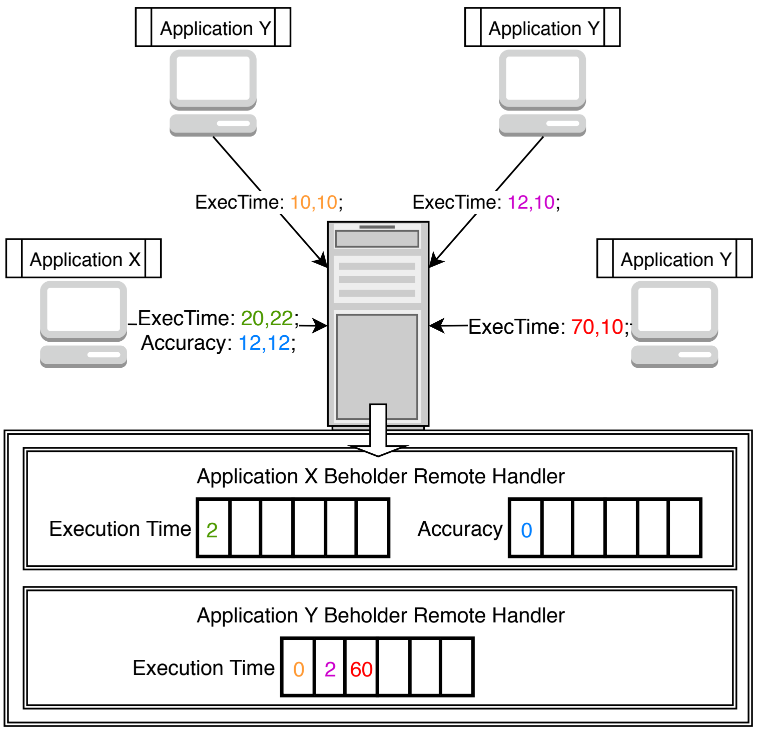


Figure 4.6: *Symbolic representation of the data collection for the first level of the CDT. The clients send MQTT messages to the server (details in Chapter 5) comprising of the observed value and estimate from the model for every (Beholder-enabled) metric. The Beholder module computes their residual (i.e. absolute difference) and fills in the respective buffers, which in this example have six slots. The buffer is the subsequence-size ν on which the ICI CDT works. The Beholder monitors two metrics for Application X in the picture, while just one for Application Y.*

change, instead of having to deal with the delay of waiting for the result of the first level test from all the metrics. Moreover, we are agnostic to which metric is actually deviating from its estimate in the model.

Assumption As presented in Subsection 2.3.5, a strong assumption, crucial for the significance of the test itself, is the stationarity of the observed behavior used for the test configuration. Meaning that the training phase has to be carried out when the application is behaving "correctly", without anomalies. During such stage, the test learns which is the acceptable range for the metric values. Obviously, outliers during the training phase mislead the test in considering such irregularities acceptable and increase the detection delay of changes.

Test parameters There are three parameters exposed to the user which influence the detection delay and the rate of false positives and false negatives. We already mentioned that the size ν of the subsequences, i.e. the metric buffers, on which the ICI CDT operates can be set. There is also the number of subsequences used for the training of this first level test. Last but not least is the Γ Confidence Interval amplification parameter, which directly impacts on the recognition delay (Equation 2.2).

Algorithm 3 Detection Layer logic pseudocode (for every remote application handler)

```
1: procedure NEW_OBSERVATION($message$)    ▷ Every time a new observation from a client arrives
2:     if $message\_coming\_from\_blacklisted\_client$ then    ▷ This step will be be clarified later on
3:         $return$    ▷ Discard observation
4:     Compute the residuals for every metric in $message$ and insert them in the respective buffers.
5:     $change\_detected \leftarrow false$    ▷ Initialize a boolean variable to false
6:     for $i \leftarrow metric\_1, all\_the\_metrics$ do    ▷ Loop through every beholder-monitored metric
7:         if $buffer\_full(i)$ then    ▷ If buffer for metric $i$ is full
8:             $change\_detected \leftarrow perform\_ici\_cdt(i)$    ▷ Perform CDT on buffer of metric $i$
9:             if $change\_detected$ then    ▷ If a change has been detected
10:                $timestamps \leftarrow save\_timestamps()$    ▷ Save the timestamps for the first and last
                                                              element of the buffer (subsequence)
11:                $empty\_all\_buffers()$    ▷ Empty all the metric buffers
12:                $break$    ▷ Break from the for cycle, no matter which metric deviated
13:            $empty\_buffer(i)$    ▷ Empty the buffer for metric $i$
14:    if $change\_detected$ then    ▷ If a change has been detected, no matter on which metric
15:        $go\_to\_second\_level\_test(timestamps)$    ▷ Go to Validation Layer
16:    else
17:        $return$ ▷ Go on, every metric is behaving according to the expectations of the current model
```

4.4 Second Level Test - Validation Layer

The Validation Layer is activated after a detection raised by the detection layer. The purpose of this layer is to validate the prospective nonstationarity by means of a hypothesis test. The objective of this test is to compare the behavior of the target application before and after the hypothetical change pinpointed by the first level searching for a confirmation, or a refutation, of the anomaly. Such a hierarchical CDT significantly reduces the number of false positives and the detection delays, since the validation layer allows for tuning the first level test to provide prompter detections.

Algorithm 4 outlines the logic underlying the second level test for every single application (inside every Beholder Remote Application Handler) and it is the pseudocode counter-part of Figure 4.8.

Data collection With respect to the detection test which collects the observations coming directly from the running clients at runtime, in this second level test we gather the information from the storage since we need to retrieve "past" data (line 6 in Algorithm 4). One of the many purposes of the Agorà module is to save in the persistent storage, which in our case is implemented using the Cassandra DBMS (Section 3.4), the log of all the observations collected from every client, with the corresponding timestamp, estimates from the model and actual monitored values; this is saved in the so-called "trace" table. When the first level test detects a change, we save the timestamp of such detection. But, as explained in Subsection 2.3.5, the accuracy with which we are able to pinpoint the time of change is $\pm\nu/2$, ν being the size of the subsequences in which the incoming observations are grouped. This is to say that we just know the time-window in which the anomaly is detected. With the hypothesis test, we are interested in comparing whether the behavior of the population before the change is different from that of the population after the change.

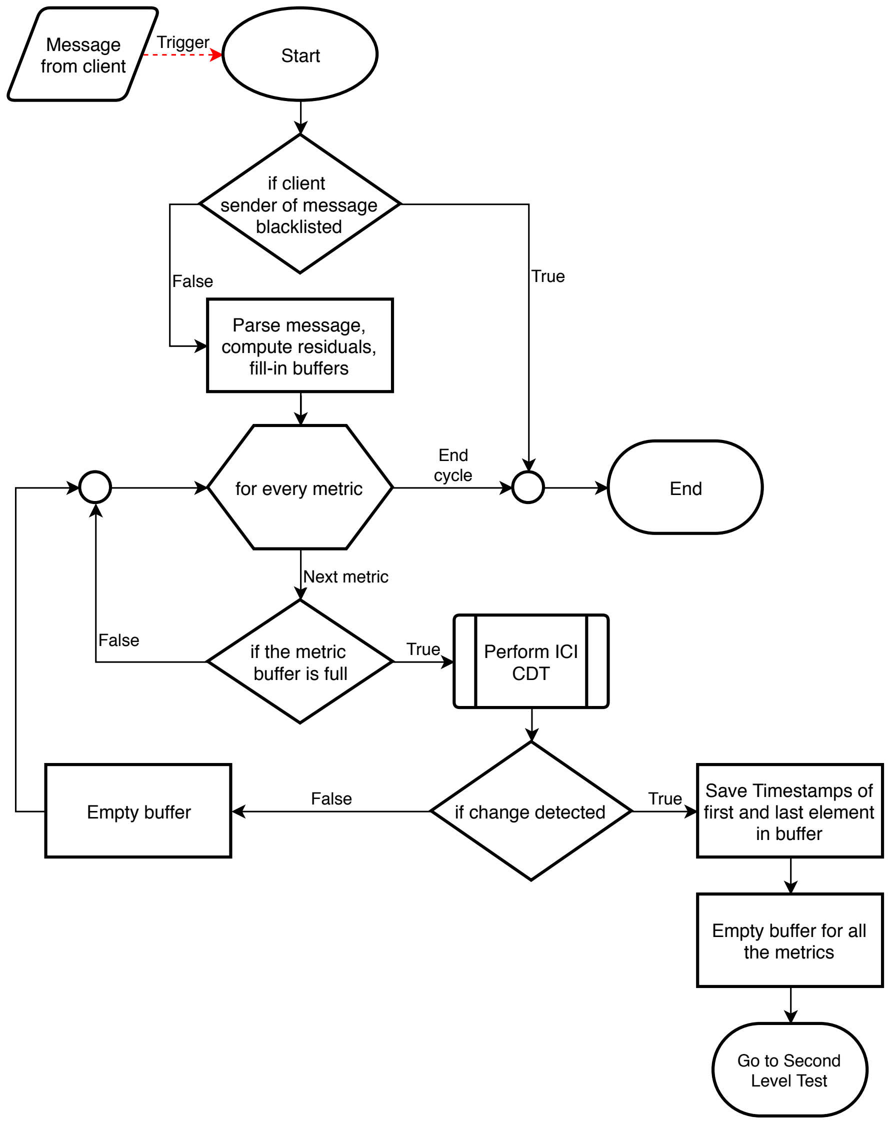

Message from client
Trigger
Start
if client sender of message blacklisted
False
True
Parse message, compute residuals, fill-in buffers
for every metric
End cycle
End
Next metric
False
if the metric buffer is full
True
Perform ICI CDT
Empty buffer
False
if change detected
True
Save Timestamps of first and last element in buffer
Empty buffer for all the metrics
Go to Second Level Test

Figure 4.7: *Flowchart presenting the logic behind the "`new_observation`" method, which is where the first level test fits. The concept of a blacklisted client will be explained later on.*

Residuals computation As for the first level test we compute and work with the residuals, i.e. the absolute difference between the estimates and the observed data for the metrics. The only difference is that here the Beholder module has to gather the observations offline from the trace table, instead of receiving them online directly from the clients; once in possession of these, it has to divide them in two groups, before and after the change, while discarding those belonging to the window in which the change has been detected.

Test granularity A difference with respect to the first level test is in the granularity of the gathered data; while in the latter the is no client-distinction, meaning that data are collected at run-time from all the nodes running the target application under analysis, in this second level the hypothesis test is carried out client-wise; this corresponds to the `for` cycle at line 3 in Algorithm 4. If there are N nodes running the same application, then N hypothesis tests will be performed. This allows understanding whether the detected change interests all the clients. Moreover, if the application observes more than one metric, at most we perform a hypothesis test for each of these metrics, as shown at line 14 in Algorithm 4. As soon as the change is confirmed on a client, we do not perform the test on the rest of the remaining metrics (if any). We just check all the metrics when the change is not confirmed (negative test).

Blacklist Among the parameters we expose to the final user there is the threshold of allowed “bad”-behaving clients expressed as a percentage. The idea is that if few nodes are deviating from the model the user may be willing to avoid a re-training. It may be because these nodes are faulty, or because they are power-throttled following overheating issues. If the (user-set) majority of clients is still working correctly according to the model, then we keep the current model.

To better manage such situations we decided to introduce a blacklist, which contains the “id” of all the bad clients. Please notice that the purpose of the blacklist is to keep track of the deviating clients only when the overall test rejects the change. Meaning that if the second level test just finds few bad nodes, whose number is under the user acceptance threshold, these will be inserted in the blacklist and the change will be rejected. The blacklisted clients will not participate in the future collection of data for the following first level test (lines 2 and 3 in Algorithm 3); this is because we do not want to be warned again about their -now- acknowledged anomalous behavior. In case another change is detected among the non-blacklisted clients, then we empty the blacklist before entering the second level test, so as to re-consider all the clients and re-compute the updated percentage of irregular clients (line 2 in Algorithm 4). Let us enforce the fact that, if the second level test confirms the change because all the clients are bad, or because their number is anyway above the threshold, then the model re-training is issued and the blacklist is not used at all.

Waiting for observations problem The hypothesis test relies on the assumption that the two compared populations have a normal distribution. In this situation, we do not know if the population displays a normal distribution. However, with a large sample size, we know from the Central Limit Theorem (CLT) that the sampling distribution of the population is distributed normally. It can happen that the first level test is carried out fast enough to enter this second test without enough samples for the population “after” the prospective change.

For such reason we had to implement a wait mechanism, outlined between line 4 and line 12 in Algorithm 4, to allow for other observations to arrive. We provide the user with the ability to set a wait-time between the queries to gather observations from the trace table, in the hope that enough observations have been collected, by then. Moreover, we introduced a customizable timeout after which this wait and query loop is stopped. When the timeout is up the client is treated as a bad one. This is because we assume the user has a rough idea of the frequency with which his/her application returns results, and then sets the parameters accordingly (Figure 4.8).

Designated Hypothesis Test A statistical hypothesis test is a method of statistical inference in which two data sets are compared. Since the size of the population before the change

Algorithm 4 Validation Layer logic pseudocode (for every remote application handler)

```
 1: procedure SECOND_LEVEL_TEST
 2:     $blacklist.clear()$    ▷ Restart from scratch and re-consider all the clients which may have been blacklisted in previous runs
 3:     for $i \leftarrow client\_1,\ all\_the\_clients$ do    ▷ Loop through every client running the application
 4:         $enough\_observations \leftarrow false$    ▷ Initialize boolean to false
 5:         while $!enough\_observation$ && $!time\_out$ do    ▷ Loop until either there are enough observations to allow the hypothesis test or the wait-time is out
 6:             $get\_observations\_from\_trace\_for\_client(i)$
 7:             Parse the observations, compute the residuals and divide them in the two populations before and after the change.
 8:             if both the two populations have enough observations then
 9:                 $enough\_observations \leftarrow true$
10:         if $!enough\_observations$ then    ▷ If we exited the While cycle for a time-out
11:             $bad\_clients\_list.emplace(i)$    ▷ We consider the current client as anomalous
12:             $continue$    ▷ Go to the next client in the for cycle
13:         $change\_confirmed \leftarrow false$    ▷ Initialize a boolean variable to false
14:         for $j \leftarrow metric\_1, all\_the\_metrics$ do    ▷ Loop through every beholder-monitored metric
15:             $change\_confirmed \leftarrow perform\_hypothesis\_test(j)$
16:             if $change\_confirmed$ then    ▷ If the change has been confirmed
17:                 $break$    ▷ Break from the for cycle, no matter which metric deviated
18:         if $change\_confirmed$ then    ▷ If the change has been confirmed, no matter on which metric
19:             $bad\_clients\_list.emplace(i)$    ▷ We consider the current client as anomalous
20:         else
21:             $good\_clients\_list.emplace(i)$    ▷ We consider the current client as aligned to the model
```

is, very likely, different from the one of the population after the change, we chose to use Welch's t-test, also known as unequal variances t-test; it is a two-sample location test which is used to test the hypothesis that two populations have equal means. Welch's t-test is an adaptation of Student's t-test, but while the latter assumes that the two populations have normal distributions with equal variances, Welch's t-test is designed for unequal variances and/or unequal sample sizes, even though the assumption of normality is maintained. Furthermore, the power of Welch's t-test comes close to that of Student's t-test, even when the population variances are equal and sample sizes are balanced [66], and for this reason it is not recommended to pre-test for equal variances and then choose between Student's t-test or Welch's t-test [74]. These tests are often referred to as "unpaired" or "independent samples" t-tests, as they are typically applied when the statistical units underlying the two samples being compared are non-overlapping. The point is that we cannot assume any correlation between different nodes running the same application; they could be processing different chunks of unrelated data.

Welch's t-test defines the statistic t by the following formula:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}} \tag{4.1}$$

where $\overline{X}_1$ ($\overline{X}_2$), s_1^2 (s_2^2) and N_1 (N_2) are the first (second) population sample mean, sample variance and sample size, respectively. The degrees of freedom ν associated with this variance estimate is approximated using the Welch-Satterthwaite equation:

$$\nu = \frac{(\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2})^2}{\frac{s_1^4}{N_1^2 \nu_1} + \frac{s_2^4}{N_2^2 \nu_2}} \tag{4.2}$$

where $\nu_1 = N_1 - 1$ and $\nu_2 = N_2 - 1$ are the degrees of freedom associated respectively with the first and second population variance estimate. Note that this is one of the rare situations where the ν parameter is a real number and not an integer value. Once t and ν have been computed, these statistics can be used with the t-distribution to test the null hypotheses that the two population means are equal (no change); a two-tailed test is applied (given that we are not interested in the direction of the change). The test configuration, in terms of hypothesis, is the following:

$$\begin{aligned} H_0 &: \mu_1 = \mu_2 \\ H_1 &: \mu_1 \neq \mu_2 \end{aligned} \tag{4.3}$$

where the Null hypothesis H_0 indicates that there is no difference between the means of the two populations, meaning that there is no change, while the Alternative hypothesis H_1 states the opposite and confirms the change. The null hypothesis is rejected, in favor of the alternative hypothesis, if and only if the p-value is less than the (user-set) significance level threshold α.

For the actual implementation of the test we rely on the C++ library "Boost Math Toolkit" [5] to compute the p-value, given the t and ν parameters. We decided to use this library because, in addition to being well-tested and supported, it fully supports non-integer degrees of freedom, while other statistical packages truncate the effective degrees of freedom to an integer value (this is also necessary when using lookup tables). This can make a difference, since when ν is small the Welch-Satterthwaite approximation may be a significant source of error [5].

Test parameters For this second level test, the user can set the significance level α of the hypothesis test. Moreover, as mentioned before, they can set the frequency-check between queries to the trace to wait for new observations and the maximum timeout period. Last but not least, they can specify the minimum number of observations which allows the hypothesis test. The latter is then carried-out only when both the populations before and after the change are at least as big as requested. Otherwise, the waiting process starts.

Bonferroni Correction The Bonferroni correction is one of several methods used to counteract the problem of multiple comparisons. In statistics, the multiple comparisons or multiple testing problem occurs when multiple statistical tests are considered simultaneously: the more inferences are made, the more likely erroneous inferences are to occur. If multiple hypotheses are tested, the chance of a rare event increases, and therefore, the likelihood of incorrectly rejecting a null hypothesis (also known as false positives or Type I errors) increases. For example, if one test is performed at the 5% level and the corresponding null hypothesis is true, there is only a 5% chance of incorrectly rejecting the null hypothesis. However, if 100 tests are conducted and all corresponding null hypotheses are true, the expected number of incorrect rejections is 5.

Techniques have been developed to prevent the inflation of false positive rates and non-coverage rates that occur with multiple statistical tests. The most conservative method, which is free of dependence and distributional assumptions, is the Bonferroni correction, according to which, in order to retain a prescribed family-wise error rate α in an analysis involving more than one comparison, the error rate for each comparison must be more stringent than α. In particular, if m is the total number hypotheses tested, then the Bonferroni correction [56] would test each individual hypothesis at α/m. The correction comes at the cost of increasing the probability of producing false negatives, i.e. reducing statistical power.

We employ this technique, and since the hypothesis test is carried out client-wise, we divide the significance level α by the number of active clients running the application under analysis. This corrected version of α is then used for every test.

Algorithm 5 Possible outcomes of the hierarchical CDT test (for every remote application handler)

```
1: procedure SECOND_LEVEL_TEST
2:     if $(bad\_clients\_list.size()/(bad\_clients\_list.size() + good\_clients\_list.size())) * 100 < bad\_clients\_threshold$ then    ▷ If the % of bad clients is under the allowed threshold
3:         for $i \leftarrow client\_1,\ all\_the\_bad\_clients$ do    ▷ Loop through every client in the list of bad clients
4:             $blacklist.emplace(i)$    ▷ Enqueue the bad client in the blacklist
5:         Reset first level ICI test to just its initial training phase    ▷ Change rejected
6:     else    ▷ If the % of bad clients is above the allowed threshold
7:         Send re-training command to Agorà    ▷ Change confirmed
8:         Reset controller    ▷ Reset the Beholder manager for the application, including the ICI test. Restart from scratch
```

4.4.1 Overall Hierarchical Test Outcomes

The outcome of the second level test is composed of two lists of clients, containing respectively those nodes who are deviating from the model and those for which the model is still valid. Once in possession of these two lists, the percentage of "bad" clients is computed with respect to the total number of clients. Algorithm 5 presents the pseudocode of the overall hierarchical CDT outcomes; while Figure 4.9 summarizes the workflow of the implemented hierarchical CDT. In particular, there are two possible scenarios:

- If there are no bad clients, or their percentage is lower than the user-set acceptable threshold, then the detected change is rejected. If present, the irregular clients are blacklisted and will not participate in the subsequent collection of data for the next run of the first level test. The ICI test is reset to its training phase since it detected an overall refuted anomaly.
- When there are no good clients, or the percentage of bad ones is above the allowed threshold, the change is confirmed. In such cases the blacklist is not used at all; the model is not sound anymore, so the Beholder triggers a re-training, carried out by the Agorà module. The Beholder controller for the current application is reset, and it is going to resume its task of model-checker once a new model is received for the specific application.

4.4.2 Effect Size Evaluation

As stated above, key assumption for the statistical soundness of the whole change detection process is that the collected samples are independent and identically distributed (i.i.d). This is also a requirement for the functioning of the ICI CDT (Subsection 2.3.5). Should this assumption be broken, we introduced a further check to allow the use of the proposed tool also in the face of a not strictly i.i.d. situation, even though in such cases the significance of the delivered results has to be carefully evaluated. During the first batch of experiments we run into a problem, which invalidated this underlying applicability condition. In particular, the application with which we were running the experiments was generating, theoretically, i.i.d. observations. In reality, the dynamic CPU frequency scaling was introducing dependencies between samples, since the execution time that we were measuring was obviously dependent on the varying frequency.

During the experiments, we simulated a change on few nodes out of all the available, and we expected the first test to be triggered, but the second level test was supposed to reject an overall change if the deviating nodes were less than the user-set allowed threshold. Unfortunately, that was not the case. We found out that actually, the main source of concerns was in the behavior of the second level test. This is because the first level test performs at its beginning a self-training to discover the range of the acceptable variation of the monitored metric(s). Thus, if the frequency of the processor already varies during that (assumed-stationary) phase, the detection layer is automatically able to cope with such underlying irregularity. It was the second layer test which, on the contrary, was not able to adapt to this unstable behavior; being it entirely based on the underlying statistical significance, it was able to spot the (albeit small) differences in performance of the execution time introduced by the continuous frequency variation, thus producing what, according to us, were false positives.

Let us stress the fact that this is a corner case which should be avoided, if possible, since it represents a misuse of the tool, given that the i.i.d. assumption on the data generating process is invalidated. We re-run all the experiments by setting a CPU governor whose policy is to keep the frequency fixed. Indeed, the best practices tell us to disable any possible form of CPU frequency scaling in order to carry out a more stable experiment, above all in contexts like ours, in which we are interested in monitoring the stability of a process, more than the raw performance. Thus, technologies like Intel® Turbo Boost, Intel® EIST (Enhanced Intel SpeedStep) and similar should be disabled in order to avoid dynamic frequency scaling (also known as CPU throttling), which is a technique in computer architecture whereby the frequency of a microprocessor can be automatically adjusted "on the fly" depending on the actual needs, to conserve power and reduce the amount of heat generated by the chip. Anyway, we decided to offer the possibility to also take into account the actual effect size of the detected variation, for the cases in which the hypothesis test confirms the change. This comes at the cost of relaxing the statistical robustness of the analysis. The idea is that a hypothesis test analyzes the statistical significance of the difference between the two compared distributions, but it does not take into account the practical significance of their difference.

In statistics, an effect size is a quantitative measure of the magnitude of a phenomenon, which can complement statistical hypothesis testing. For most types of effect size, a larger absolute value always indicates a stronger effect [47]. Sample-based effect sizes are distinguished from test statistics used in hypothesis testing, in that they estimate the strength (magnitude) of, for example, an apparent relationship, rather than assigning a significance level reflecting whether the magnitude of the relationship observed could be due to chance. The effect size does not directly determine the significance level, or vice versa. Given a sufficiently large sample size, a non-null statistical comparison will always show a statistically significant result unless the population effect size is exactly zero (and even there it will show statistical significance at the rate of the Type I error used). In our scenario, if this happens it is because the i.i.d. assumption was broken. For example, a sample Pearson correlation coefficient (a measure of the linear correlation between two variables) of 0.01 is statistically significant if the sample size is 1000. Reporting only the significant p-value from this analysis could be misleading if a correlation of 0.01 is too small to be of interest in a particular application.

We are interested in evaluating the effect sizes based on the difference between the means of the populations before and after the detected change. An effect size θ based on means usually considers the standardized mean difference between two populations:

$$\theta = \frac{\mu_1 - \mu_2}{\sigma} \tag{4.4}$$

where μ_1 is the mean for one population, μ_2 is the mean for the other population, and σ is a standard deviation based on either or both populations. This form for the effect size resembles the computation for a t-test statistic, with the critical difference that the t-test statistic includes a factor of $\sqrt{n}$. This means that for a given effect size, the significance level increases with the sample size. Unlike the t-test statistic, the effect size aims to estimate a population parameter and is not affected by the sample size. In the practical setting, the population values are typically not known and must be estimated from sample statistics. The several versions of effect sizes based on means differ with respect to which statistics are used.

We used Cohen's d [36], which is defined as the difference between two means divided by a standard deviation for the data:

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s}. \tag{4.5}$$

Jacob Cohen defined s, the pooled standard deviation, as:

$$s = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n1 + n2 - 2}}, \tag{4.6}$$

where n_1 (n_2) is the first (second) sample size, while the variance for one of the groups is defined as

$$s_1^2 = \frac{1}{n_1 - 1}\sum_{i=1}^{n_1}(x_{1,i} - \bar{x}_1)^2, \tag{4.7}$$

and similar for the other group. Table 4.1 contains descriptors for magnitudes of d = 0.01 to 2.0, as initially suggested by Cohen and expanded by Sawilowsky [68].

We provide the option to enable or disable this effect size check and to set the threshold d. When such check is enabled, if the hypothesis test rejects the null hypothesis and confirms the change, then the d statistic is computed and compared with the user-threshold. Only when the d is above such threshold the change is actually confirmed.

Effect Size	d
Very small	0.01
Small	0.20
Medium	0.50
Large	0.80
Very large	1.20
Huge	2.0

Table 4.1: *Correlation between Cohen's* d *and effect size.*

It is important to notice that, although we introduced this further test, we did not use it for the results reported in Chapter 6.

4.5 Conclusions

In this chapter, we addressed all the methodological choices at the base of the contributions made to the framework. In particular, we started from the additional features that allow a much better integration of the proposed workflow. That is to say, the possibility to synchronize the application with the remote module Agorà, the option to investigate on the current status of the application knowledge, whether there is a model or if the training phase is currently in progress. We introduced the management of the new error monitor to automatically disable the error computation during the production phase. Finally, we presented the structure of the hierarchical CDT, core innovation of our approach. In particular, the first level is a CDT which continuously monitors the execution by evaluating the collective behavior of the running nodes, while the second level is in charge of confirming or rejecting a detected anomaly by carrying out a client-wise analysis. As a further backup layer, we proposed an evaluation of the practical effect size of the change, to be used in those situations which break the otherwise required i.i.d. assumption. In the next chapter, we will take care of the implementation details underlying all these contributions.

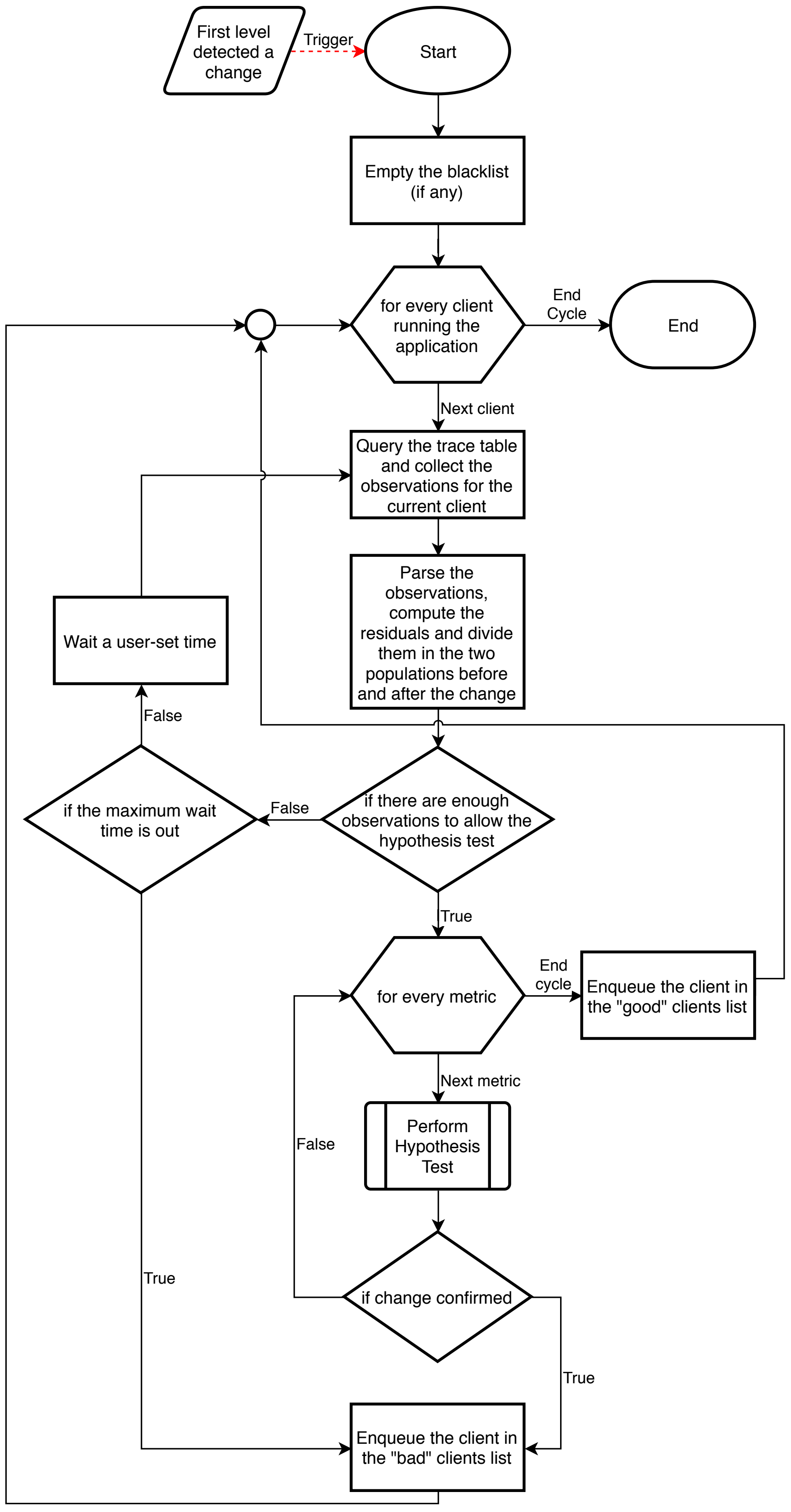

First level detected a change
Trigger
Start
Empty the blacklist (if any)
for every client running the application
End Cycle
End
Next client
Query the trace table and collect the observations for the current client
Parse the observations, compute the residuals and divide them in the two populations before and after the change
Wait a user-set time
False
if the maximum wait time is out
False
if there are enough observations to allow the hypothesis test
True
for every metric
End cycle
Enqueue the client in the "good" clients list
Next metric
Perform Hypothesis Test
False
True
if change confirmed
True
Enqueue the client in the "bad" clients list

Figure 4.8: *Flowchart presenting the logic behind the second level test.*

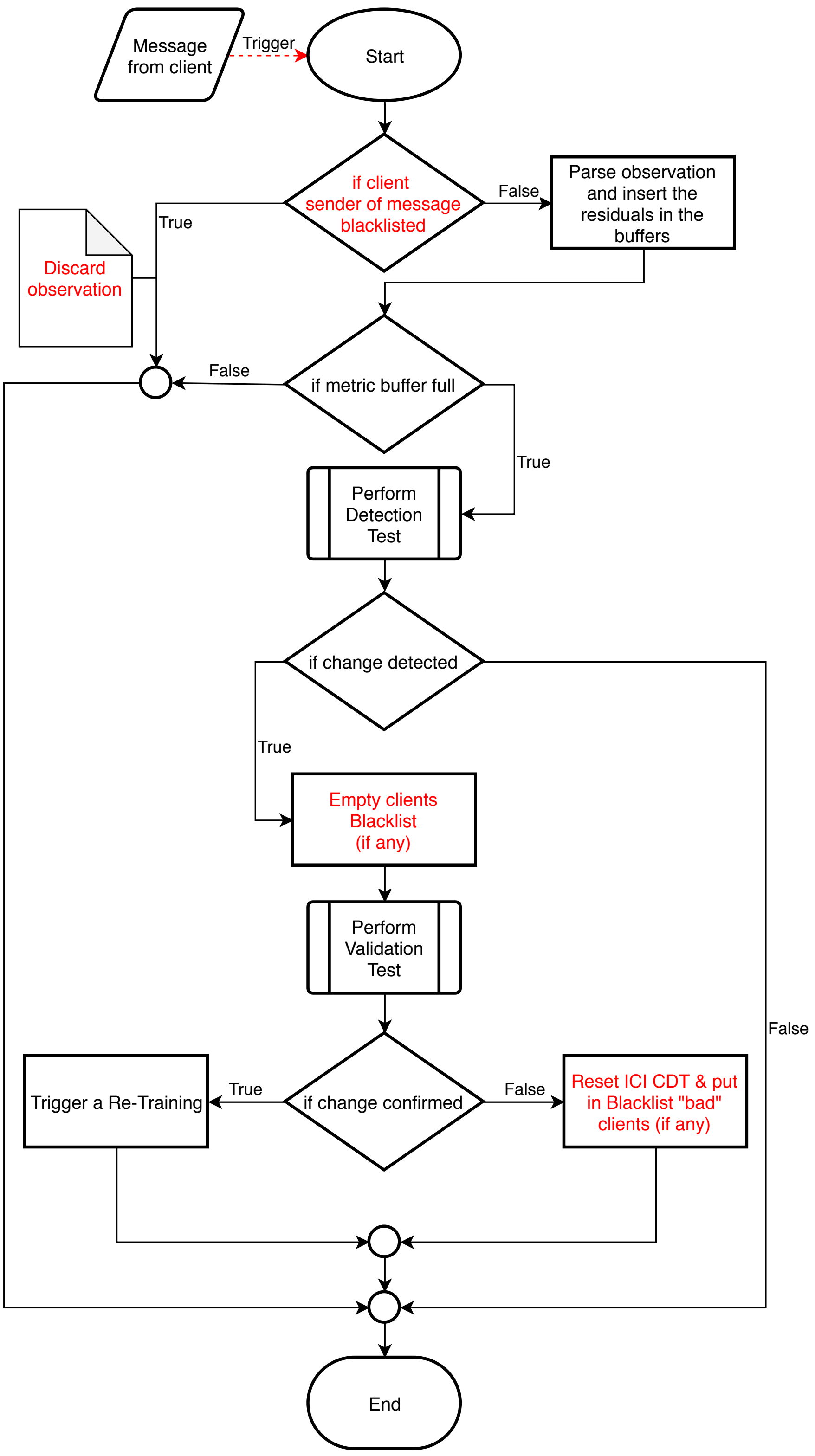


Figure 4.9: *Flowchart presenting the logic behind the hierarchical test we adopt, in which the interaction between the two layers and the possible final outcomes are summarized. Highlighted is the functioning of the blacklist.*

CHAPTER 5

Beholder Module

While the methodological choices underlying the functioning of the new module have been taken care of in Chapter 4, in this chapter we delve deeper into the technical aspects of the implementation and integration of the Beholder module inside the preexisting mARGOt framework. We present the interactions between the different components through the use of sequence diagrams, to better understand the roles of every actor involved in the workflow; also highlighted are the contents of the messages which are exchanged among the different parties.

5.1 Module Placing: Server-Side

The Beholder module has to collect observations coming from all the clients to perform its task of model-checker by means of a Change Detection Test (CDT); thus, it needs to possess an overall view of the distributed architecture. The Beholder is a process placed on a central server; and given that its ultimate purpose is to trigger, when needed, a re-training of the model, task carried out by the Agorà module, which already resides on a server, we considered reasonable to place the Beholder besides Agorà. Since the Beholder offers an optional functionality, it is the final user who decides whether to launch it or not; obviously mARGOt is still able to perform its own purpose without it, in such case foregoing the perks introduced by the Beholder. The increase in computation overhead due to the addition of a new module and its communication channels is negligible, since we deploy it on a server which is assumed not to be a bottleneck in our situation, above all when considering an HPC facility.

Figure 5.1 shows the structural changes brought with the introduction of the Beholder in the server structure; communication-wise, the Beholder has to interact with Agorà to trigger the re-training, with the client applications in order to perform the online ICI

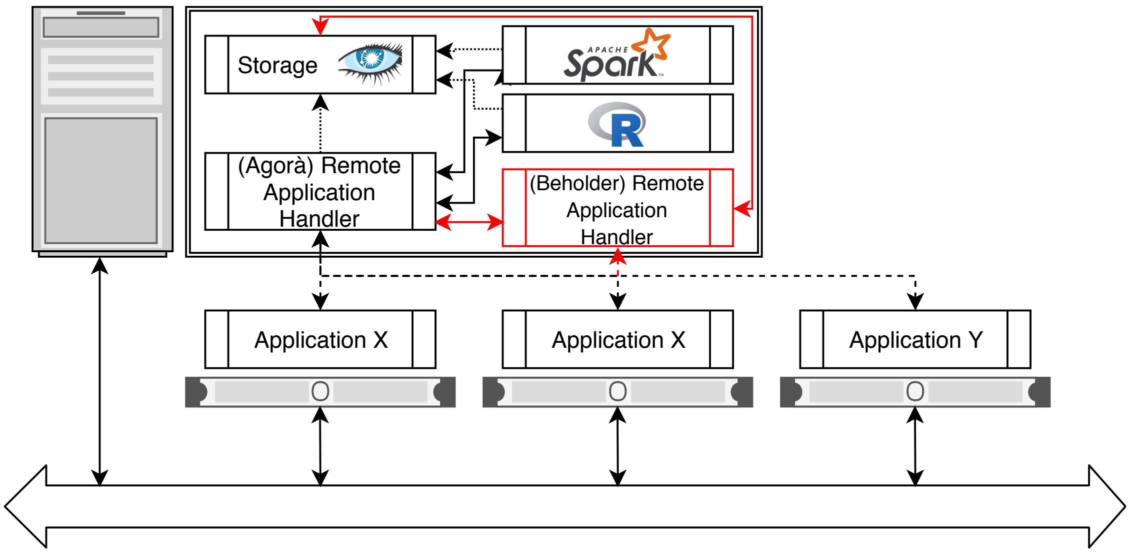

Storage
Spark
(Agorà) Remote Application Handler
(Beholder) Remote Application Handler
Application X
Application X
Application Y

Figure 5.1: *Beholder module placement, in red are the changes with respect to the "old" version strapped of the Beholder (Figure 3.14).*

Change Detection Test (Section 4.3) and eventually with the storage to collect the past data from the storage used by the Validation layer (Section 4.4).

5.2 Beholder Process

The Beholder is a process which has to be launched on a server, theoretically, the handiest location to run it is on the same server where Agorà lies, even though this is not a strict requirement. The order in which the Beholder is launched with respect to the rest of the framework is totally indifferent. Meaning that the user can launch Agorà and later on the Beholder, or vice-versa. In theory, the Beholder can also be launched while the application is already running. Obviously, in such case, the Beholder will check the model against the observed behavior only from the instant in which it is launched. We designed a pattern of interaction that is fixed and executed once, when the Beholder is started. This allows the above-mentioned flexibility of launch order options. A predetermined exchange of messages takes place, and thanks to this the Beholder inquires into the present status of the whole framework and synchronizes itself, preparing to carry out its task.

5.2.1 Remote Application Handler

In the Beholder module, similarly to Agorà, there is a data structure for every application, the "Remote Application Handler". It is in charge of describing the application-specific status. There is a never-ending worker process which listens for incoming messages and processes them by using the handler of the corresponding sender application. We decided to avoid the use of any permanent storage for the Beholder. Meaning that once its process is stopped, everything is lost. This is because we did not want to trigger any re-training, a very onerous activity which is only carried out as last option, basing such decision on possibly no longer sound data, belonging to checks partially performed on

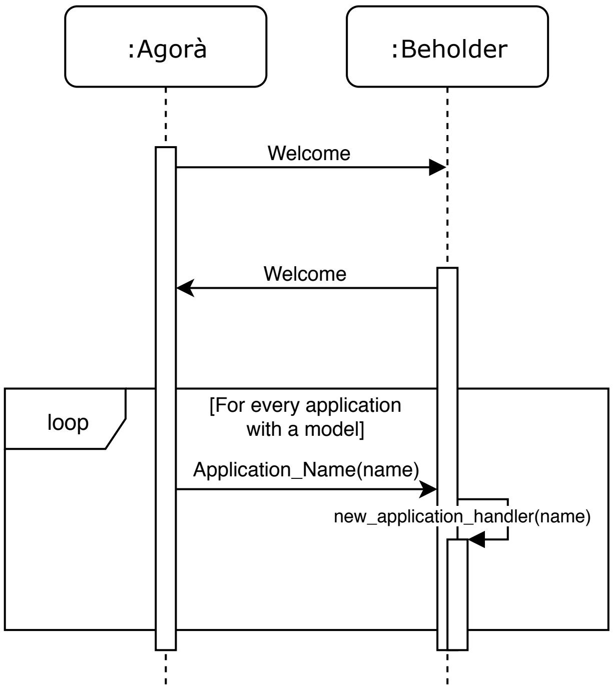

:Agorà
:Beholder
Welcome
Welcome
loop
[For every application with a model]
Application_Name(name)
new_application_handler(name)

Figure 5.2: *Sequence diagram presenting the initial knowledge sharing and synchronization between Agorà and the Beholder; case in which the former is launched first. For every application for which Agorà already has a model, the Beholder creates an application handler. Please note that the interaction in the picture is symbolic; messages are, in fact, sent to the MQTT broker which forwards them to the specific topic subscribers.*

previous models and restored across launches of the Beholder process. This keeps all the workflow of the Beholder simpler and at the same time more robust with respect to errors, but it also requires an overview of the whole framework status every time the Beholder process is launched.

5.2.2 Lifecycle with respect to Agorà

First and foremost, let us remind that the purpose of the Beholder is to monitor the application model used by the autotuner and evaluate its correctness at runtime. That being said, it is straightforward to understand why we are only interested in analyzing applications which are already being autotuned according to a model. We saw how Agorà allows for a model training at runtime by guiding an online DSE; we are not interested in monitoring such phase with the Beholder, since in this case the model still has to be built. The approach we adopted to share the knowledge about the framework status is based on an initial synchronization between Agorà and the Beholder; this is because Agorà is aware of the applications for which there is a model already.

In order to carry out such coordination between the Beholder and Agorà, we decided to synchronize the two processes, and the Beholder, being optional, is the enforcer of such task by means of an exchange of MQTT messages. The order of such messages depends on which process is launched first. The fixed pattern in their behavior is that both the Beholder and Agorà, as soon as they are launched, send a “Welcome” message to whoever listens to the corresponding topics. Thus, when Agorá is launched first and

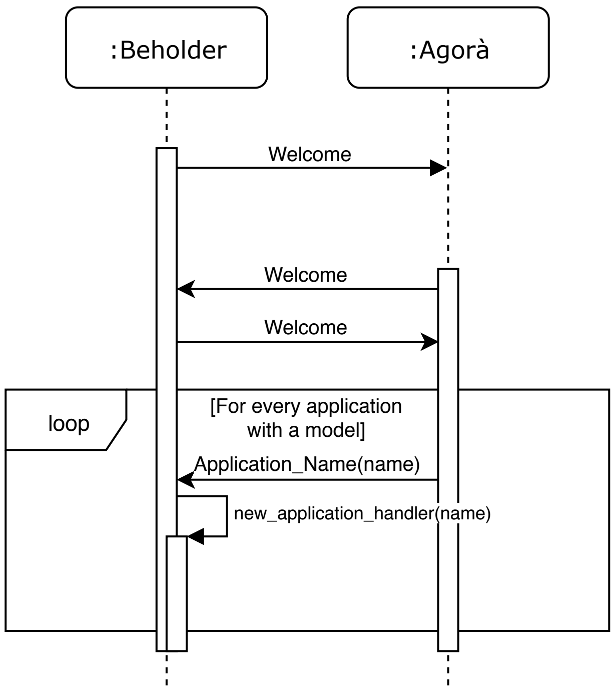

:Beholder
:Agorà
Welcome
Welcome
Welcome
loop
[For every application with a model]
Application_Name(name)
new_application_handler(name)

Figure 5.3: *Sequence diagram presenting the initial knowledge sharing and synchronization between Agorà and the Beholder; case in which the latter is launched first. For every application for which Agorà already has a model, the Beholder creates an application handler. Please note that the interaction in the picture is symbolic; messages are, in fact, sent to the MQTT broker which forwards them to the specific topic subscribers.*

sends its own welcome message, the Beholder is not there to listen for it; for this reason, once the Beholder is started, it sends its own welcome in turn. Once Agorà receives it, it will send a message destined to the Beholder for every application for which there exists a model, as shown in Figure 5.2.

The same goes for the situation in which the Beholder is started first; it will send a welcome message to Agorà, which is still offline. Once Agorà is launched, it will send its own welcome message, to which the Beholder will reply sending again, in turn, its own welcome message. Finally, as in the first scenario, Agorà will respond with as many messages as the applications with a model are (Figure 5.3).

After this initial phase, the knowledge of the Beholder is kept updated by listening for new application models as they arrive. Essentially, every time a new application is integrated in the framework, Agorà guides it to perform the required DSE. Once Agorà computes the model, the latter is broadcasted. The Beholder, as also the nodes running that specific new application, listens for such broadcast message containing the model. The clients will be interested in the content of the message, the model itself. The Beholder, on the contrary, is just interested in knowing that it can, from now on, monitor the new application, since its model exists (Figure 5.4). To summarize, once started, the Beholder inquires Agorà on which are the applications with a model already available, if any. While, for possible new applications, their broadcasted model message itself is the trigger for the Beholder.

We decided to partially bind the lifecycle of Agorà with the one of the Beholder.

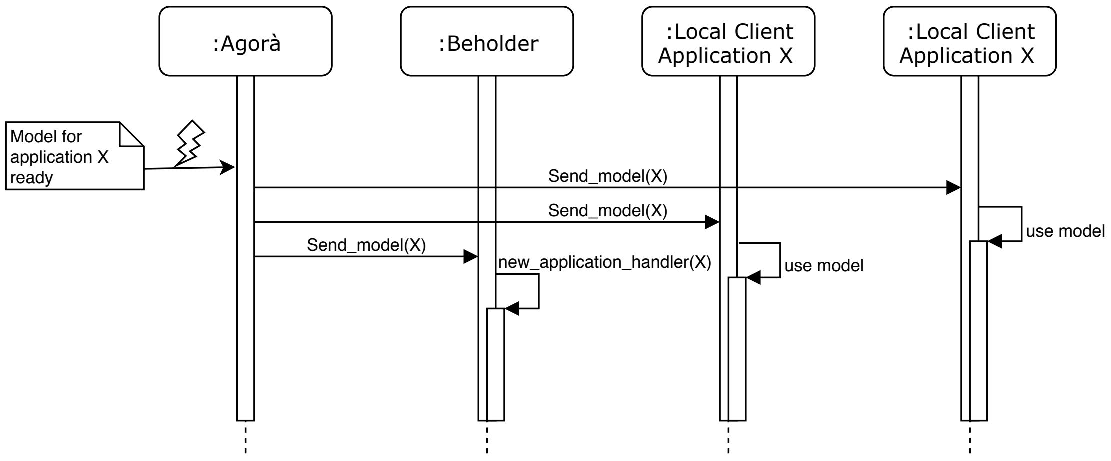

:Agorà
:Beholder
:Local Client Application X
:Local Client Application X
Model for application X ready
Send_model(X)
Send_model(X)
Send_model(X)
new_application_handler(X)
use model
use model

Figure 5.4: *Sequence diagram representing how the Beholder knowledge is kept updated after the initial knowledge synchronization with Agorà. Essentially the Beholder subscribes to the model broadcast messages. This represents the "trigger" which makes the Beholder acknowledge the presence of new applications on which to perform its analysis. Please note that the interaction in the picture is symbolic; messages are, in fact, sent to the MQTT broker which forwards them to the specific topic subscribers.*

This is to say that our approach is based on the idea of pausing the Beholder activity if Agorà should disconnect for any reason. This choice is justified by the fact that when applications are autotuned, they send the observed metrics results to Agorà, which is in charge of storing them in the trace table. The trace table is later on used by the Beholder to perform the second level of the hierarchical CDT (Section 4.4). Thus, when Agorà is offline there is no memorization, hence there is no point in letting the Beholder work in such case. Besides, Agorà is the module which takes care of the (re)training request issued by the Beholder.

As far as the Agorà disconnection detection is concerned, we exploit the Last Will and Testament (LWT) offered by the MQTT protocol, briefly introduced in Section 3.3. Essentially the Beholder is warned, if the Agorà process terminates, by the MQTT broker. When this happens, all the Beholder application handlers are paused, as shown in Figure 5.5.

We will focus on the details of this mechanism later on; until then, we can anticipate that this is put in practice by saving the present "status" of the handler, which is an indicator of the current situation of such handler, in a temporary variable "previous_status", to allow for a graceful restore of the context later on, while the status is put to "DISABLED". When Agorà should come back online, its welcome message would un-pause the beholder which would re-enable its handlers by restoring their previous status.

5.2.3 Interactions with the client Application

The Beholder module receives, directly from the nodes running the applications, the messages containing the observations. Each client, at the end of a cycle, sends the just produced observations to a dispatcher of messages. This dispatcher always sends the messages to Agorà for logging purposes in the trace table; while it only sends the message to the Beholder if the client is running according to a model. As we mentioned

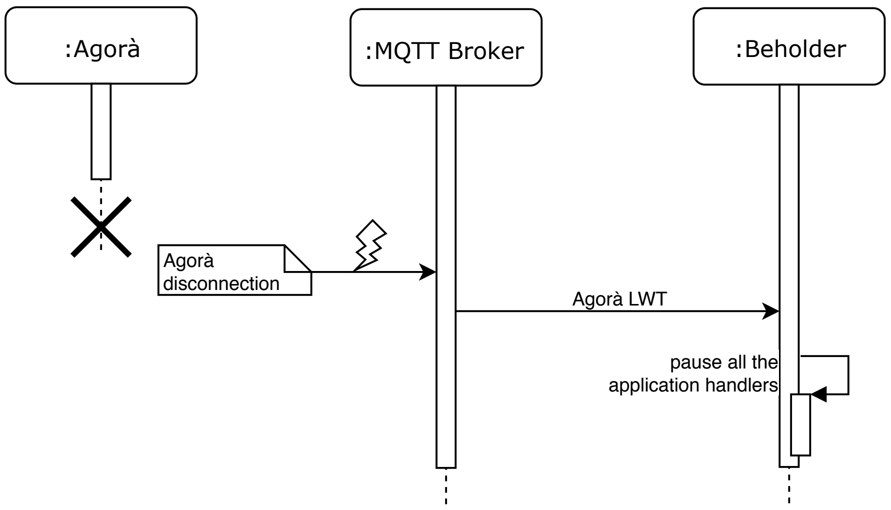


Figure 5.5: *Sequence diagram presenting how the MQTT's LWT mechanism works; here employed to warn the Beholder about possible Agorà's disconnections.*

before, the Beholder is not interested in receiving observations not coming from a model evaluation, for instance, when the client is collecting training data according to Agorà's instructions. Thus sometimes two messages are sent, one for Agorà and one for the Beholder, and other times only the one addressed to Agorà is.

In the example in Figure 5.6 the application is running according to a model, in fact, it also sends the message addressed to the Beholder. The two messages are different, thus published on two different MQTT topics. Please note that the interaction in the picture is symbolic; messages are, in fact, sent to the MQTT broker which forwards them to the specific topic subscribers. Moreover please note that, even though they are similar, the messages that the clients send to Agorà and to the Beholder are not the same. While for logging purposes we insert every available information in the trace, the message for the Beholder is stripped of the information not needed (e.g. the selected features cluster), and just contains, for every Beholder-enabled metric, the observed value and the estimate gathered from the model. The Beholder-addressed message is of the kind:

$$\begin{aligned}\langle client\_id,\ timestamp,\\ metric1\_observation,\ metric1\_estimate,\\ metric2\_observation,\ metric2\_estimate,\\ metricN\_observation,\ metricN\_estimate,\\ metric1\_name,\ metric2\_name,\ metricN\_name\rangle\end{aligned}$$

Once the Beholder is in possession of such information, it computes the absolute difference between the observation and estimate for every metric and stores that in its corresponding buffers, as explained in Section 4.3. The module works with single-precision floating-point data.

Active Clients list The Beholder keeps track of all the active clients for every application. This is because, when the second level test is performed, we weight the number of "good"

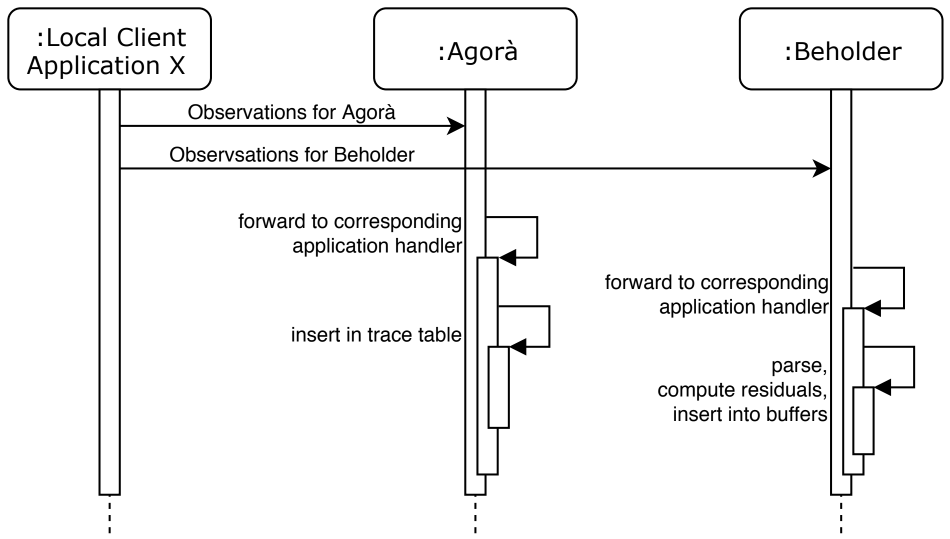

:Local Client Application X
:Agorà
:Beholder
Observations for Agorà
Observsations for Beholder
forward to corresponding application handler
insert in trace table
forward to corresponding application handler
parse, compute residuals, insert into buffers

Figure 5.6: *Sequence diagram depicting the observations transmission towards the Beholder and Agorà.*

versus “bad” clients; in this situation we want to assess the model against the running clients which triggered the detection layer in the first place, avoiding to take into account the disconnected clients. A client is added in such list once the Beholder receives the first observation from it; while the mechanism to detect the disconnection of nodes is the same used to intercept the disconnection of Agorà, i.e. the MQTT LWT. Once the Beholder receives the warning about the disconnection of a client, this is crossed out from the list of active clients.

Please note that, when a client enters the list of active clients, the timestamp of its first message is recorded. The list of active clients is, in fact, an hash-map, whose key is the client id, while the value is the timestamp of the first observation from that specific node. This is very handy when querying the trace during the second level test, to select only the part of trace following this “first-time-seen” timestamp.

Figure 5.7 shows how the data retrieved from the trace table is actually treated for the division in the two populations before and after the detected change, a necessary task to perform the second level test, as explained in Section 4.4. Let us remember that this process is carried out client-wise. The population before the change starts from the first observation recorded by the Beholder for that specific client, thus the need of the hash-map which associates the client to its first timestamp. The window of observations where the change was detected is crossed out of the analysis since we are not able to pinpoint when the change actually happened inside such subsequence.

Client list snapshot The hash-map which keeps track of the active clients running the application is an application handler-related data structure that is mutex-protected. Both the insertion/deletion of clients has to be thread-safe. This introduces a new concern which has to be taken into account. Essentially the second level test is performed on every single client running the application, as shown in Algorithm 4, through a trivial for-loop. This means that, when a change is detected by the first level CDT, we enter the second level test with a given list of clients. Then the lock is released; we will explain the locking policies later on. In such a situation, it may happen that, while performing

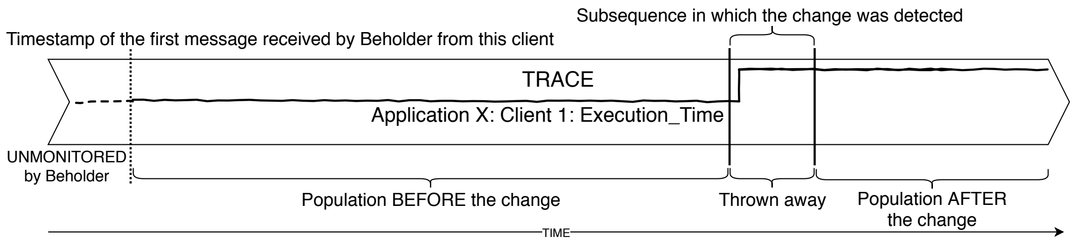

Subsequence in which the change was detected
Timestamp of the first message received by Beholder from this client
TRACE
Application X: Client 1: Execution_Time
UNMONITORED by Beholder
Population BEFORE the change
Thrown away
Population AFTER the change
TIME

Figure 5.7: *Symbolic representation of the trace, which is, essentially, a timeline. For the second level test there is the need for dividing the observations in two sets: before and after the hypothetical change detected by the first level test.*

the validation layer, few clients stop their execution and disconnect; or new ones could join.

Since we are interested in the statistic of good versus bad clients, in particular considering those clients which triggered the ICI CDT test in the first place, we save a snapshot of the current list of clients before releasing the lock and starting the second level test. Then we perform the hypothesis test on this frozen copy of the list, as shown in Figure 5.12. In this way, we can be consistent in the analysis, while at the same time keeping the real list of active clients correctly updated at runtime.

5.3 Beholder Parameters

The Beholder exposes many parameters which control the two levels of CDT presented in Chapter 4. These are server-side options, valid for all the running applications and are set to default values which should be generic enough that they require no adjustments in the majority of the cases. The settings of these knobs impacts above all the Recognition Delay (RD), the False Positive (FP) and False Negative (FN) metrics. Obviously, there may be situations in which the user may be willing to tune differently such options, according to their application and specific requirements. For the actual implementation of such parameters we rely on the well known C++ library of Boost "Program Options" [4]. Here is a list of the most important options, already mentioned in Section 4.2:

1. First Level Test related parameters:
   - `window_size`: the number of samples that fit in a single window of observations on which the ICI CDT operates;
   - `training_windows`: the number of subsequences to be used as training for the first level ICI CDT;
   - `gamma_mean`: parameter which controls the amplification factor Γ of the Confidence Interval for the mean; if greater than 1 it delays the change detection reducing the number of false positives.
   - `gamma_variance`: parameter which controls the amplification factor Γ of the Confidence Interval for the variance; if greater than 1 it delays the change detection reducing the number of false positives;
   - `variance_off`: disables the sample variance feature of the ICI CDT which will just use the sample mean feature;

2. Second Level Test related parameters:

    - `bad_clients_threshold`: the percentage of clients, for every application, that is allowed to deviate from the model. The maximum acceptable threshold;
    - `min_observations`: minimum number of observations (in the two populations before and after the change window selected in the first level test) to allow the hypothesis test;
    - `frequency_check`: frequency of the check (expressed in seconds) for new incoming observations in the trace table. The check will be carried out until either the `min_observations` number is reached or the wait time runs out according to the `timeout`;
    - `timeout`: timeout (expressed in seconds) to stop the waiting process in the second level of the hierarchical CDT for the observations to reach the `min_observations` number;
    - `alpha`: α (significance level) used in the second level hypothesis test.

Other parameters include the storage and communication options which allow the interaction with the Agorà module and the Cassandra DBMS, plus the MQTT broker address. These are needed if one wants to launch the Beholder on a different server with respect to where Agorà is running. There is also the "`number_of_threads`" parameter, which represents the number of worker threads to dedicate at the Beholder process, which will be used to manage the thread pool structure that we present below; the level of logging and the path where output files can be saved are other options. The latter enables the framework to export information needed to plot the curves of the ICI CDT, among others, with a provided Gnuplot script, to better visualize the metrics behavior. Moreover, there is the possibility to enable the effect size check and to set the corresponding Cohen's d threshold. The user can set the metrics on which the Beholder performs its checking action. As mentioned in Section 4.2, for such customization the user has a parameter in the XML configuration file. This is the only client-side option, and it works application-wise.

5.4 Managing Concurrency

We set up a system based on a thread pool, which is a software design pattern for achieving concurrency of execution in a computer program. It involves keeping multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program. By maintaining a pool of threads, the model increases performance and avoids latency in execution due to frequent creation and destruction of threads for short-lived tasks. The number of available threads has to be tuned according to the computing resources available, such as parallel processors, cores, memory, and network sockets. Note that the number of threads does not (necessarily) coincide with the number of handlers. Threads and handlers are not bound in any way. For instance, if the Beholder is currently managing N applications, i.e. checking their behavior, there will be exactly N application handlers; while M, the number of worker threads, may or may not be equal to N. It depends on the tunable `number_of_threads` Beholder parameter.

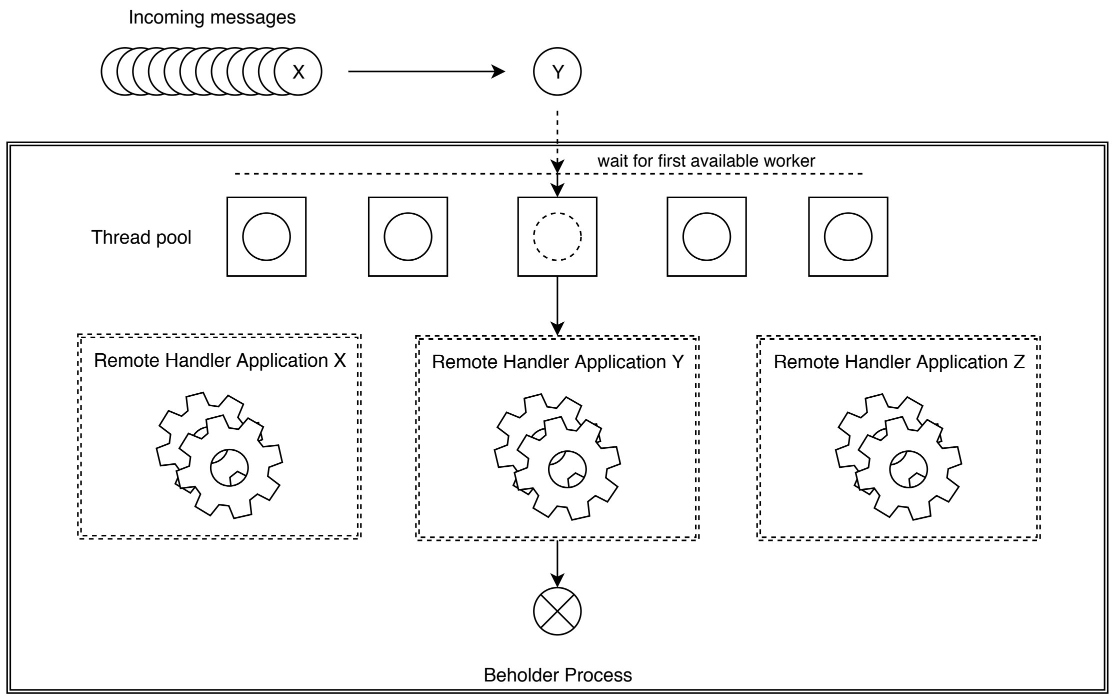

Incoming messages
X
Y
wait for first available worker
Thread pool
Remote Handler Application X
Remote Handler Application Y
Remote Handler Application Z
Beholder Process

Figure 5.8: *Outline of Beholder logic structure. Note that the number of workers (5 in the picture) is not (necessarily) the same as the number of application handlers (3 in the picture).*

Thus, when the Beholder process is started it spawns M threads. Every one of these listens for new incoming messages. Figure 5.8 presents a symbolic representation of the thread pool structure employed in the Beholder. When a message arrives a thread parses it and handles its processing. The thread which takes on a message handles it using the corresponding recipient application handler. Once the task finishes, the thread is ready to listen for another message. Let us remember that for every running application on the clients, if there is the corresponding application model, then the Beholder instantiated a logical processing engine, the remote application handler, which takes care of that application. This works application-wise: every application has its own corresponding handler in the Beholder.

Since there are multiple worker threads, when many messages arrive at the Beholder, from multiple nodes running the same application, these must be managed correctly, since they have to be processed by the unique application handler dedicated to that application. The application handler has several data structures, for instance, the metric buffers, which should not be accessed simultaneously by different threads, otherwise, the risk of "collateral damage", in return for the faster processing gained out of the parallelism, is very high. Thus the need to take actions to ensure thread-safe code, which only manipulates shared data structures in a manner that guarantees that all threads behave properly and fulfill their design specifications without unintended interaction.

The function which is executed more often in the whole Beholder module is the "`new_observation()`". It is in charge of parsing the incoming message, computing the residuals, filling in the correct buffers, and, if any of these is full, starting the first level CDT on them; should the first level detect a change, then the second level test is performed. Obviously, this method is quite complex and is at the core of the

Beholder functioning. We decided to partially mutex-protect its execution to avoid inconsistencies in its data structures when these are accessed concurrently. As soon as an observation message is received by the Beholder, it is processed by means of a worker thread of the thread pool. This thread will take care of the message until all the actions required to completely handle it have ended. The processing is delegated to the handler of the application sender of the message. Immediately after the start of the `new_observation()` function, the lock is acquired, meaning that no other thread can interfere with the computation in action. Other threads would be waiting their turn. As soon as the processing of the message finishes the lock is released.

To be more precise, the lock is acquired immediately, and it is released once:

- the observation is parsed, its residuals inserted in the buffers, but none of the latter is full, so the first level test is not performed and the function terminates;
- the observation is parsed, its residuals inserted in the buffers, and at least one of the latter is full, allowing the first level test, which however does not detect any anomaly, meaning that the observations are in-line with the model; the second test is not performed and the function terminates.

There is just another slightly different case in which we release the lock. When the first level test detects a change and this triggers the second level test. The premise is that, since the second level test retrieves data from the storage, it may be significantly slower, since interacting with the DBMS typically adds a non-negligible overhead. In such cases, keeping the lock could entail slowing down the whole processing of streaming data. Another justification for this design choice is that, as explained in Section 4.4, if there are too few samples in the population after the prospective change, there could be the need of waiting for new observations to come. This is the only situation in which we release the lock, even though the current function has not ended yet. However, we ensure the correctness of the processing by making use of a stateful application handler. We will address this matter in the following section.

5.5 Application Handler Statuses

The current situation of an application handler, and the allowed actions at any moment in time, are represented by its "`status`" variable. Here is the list of possible values for the handler `status`:

- `READY`: the handler is ready to receive new incoming observations for the collection of the necessary data to perform the ICI test. As soon as a handler is created it is set to this status;
- `COMPUTING`: the handler discards any possible new incoming observation, since it is performing the hypothesis test already. The handler could also be waiting for new observations to be received by Agorà, which keeps filling the trace if the application is still running; this happens when the hypothesis test cannot be performed for lack of observed data after the hypothetical change. The handler is set to this status when entering the possibly slow second level test after a detected change by the detection layer. When releasing the lock, we allow other threads to enter the `new_observation()` method, but as soon as they enter, if the status

```
void RemoteApplicationHandler::new_observation(const std::string&
    values)
{

  // lock the mutex to ensure a consistent global state
  std::unique_lock<std::mutex> guard(mutex);

  // check whether we can analyze the incoming payload or if we
    need
  // to discard it according to the handler status
  if ( status != ApplicationStatus::READY )
  {
    return;
  }
  ........
}
```

Figure 5.9: *Snippet of C++ code which checks for the current* `RemoteApplicationHandler` `status` *before dealing with the incoming message payload.*

is different from `READY`, they will immediately exit discarding the observation, without touching any data structure, as shown in the excerpt of code in Figure 5.9. This mechanism prevents that the queue of observations waiting to be processed grows uncontrollably. Please note that, once the second level test finishes, and we can compare the number of "good" clients, which behave according to the model, against those "bad" ones deviating from the model, then a decision must be taken (Subsection 4.4.1). Either issuing a retraining or resetting the first level ICI test. In either case, we need to re-acquire the lock to ensure consistency of the data. Thus, before actually deciding on the future of the application model, we re-lock the resources. Once the final actions have eventually been applied, then the lock is released. Figure 5.12 shows the overview of all the possible points of lock and release of the mutex inside the `new_observation()` method.

- `DISABLED`: the handler is set to this state once it receives the Agorà's LWT, to indicate that the latter is offline. The Beholder pauses the handler. The former status is saved in a `previous_status` variable, to allow for a graceful restoring of the operations once Agorà comes back online.

- `TRAINING`: the handler is set to such state once it has issued a re-training request to Agorà, at the end of the second level test. It rejects any possible incoming observation because belonging to clients which still have the old, just invalidated, model; this may happen since it is Agorà which, after having received the re-training message, has to send the new configuration to be explored to the interested nodes, effectively deleting the old model with which they were working. Such a process takes time, so if the kernel of the application is executed very quickly, the clients may still have time to send observations computed with the invalidated model. Should it happen, we want to discard these, thus the need for the check also on the Beholder's side, put in practice by using this intermediate state before setting back to READY. When a new broadcasted model message for the corresponding

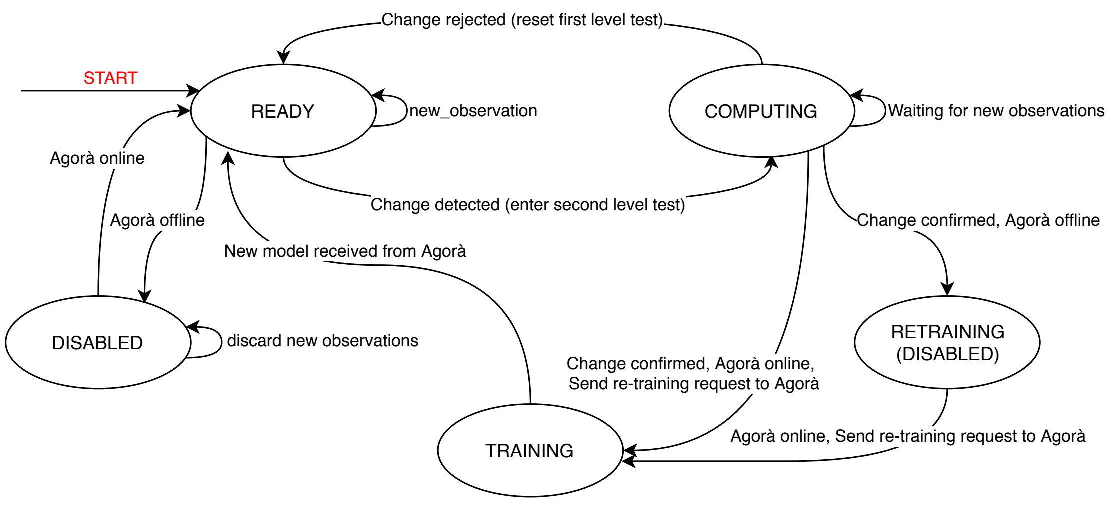

START
READY
COMPUTING
DISABLED
TRAINING
RETRAINING (DISABLED)
Change rejected (reset first level test)
new_observation
Waiting for new observations
Agorà online
Agorà offline
New model received from Agorà
Change detected (enter second level test)
Change confirmed, Agorà offline
discard new observations
Change confirmed, Agorà online, Send re-training request to Agorà
Agorà online, Send re-training request to Agorà

Figure 5.10: *FSA describing the behavior of the* `status` *variable, its possible values are represented by the states, while the arcs show the conditions to transition from one state to another. The representation is symbolic; the RETRAINING is actually a value that only the* `previous_status` *can assume. Please note that we did not flag any state as terminal, since, in theory, the process never ends; it is user-terminated, otherwise the handler is never destroyed. Thus at any state the process can be manually interrupted.*

application is received, the handler is set back to READY.

- `RETRAINING`: status to which the handler is set when, at the end of the second level test, it determines the need for a new model, but Agorà is offline, hence there is no point in issuing a re-training request that would go undetected. This state can only be found in the `previous_status` variable while the `status` variable is DISABLED. When Agorà comes back online, the re-training message is issued and the status switches to TRAINING.

Figure 5.10 outlines the above-mentioned behavior of the `status` variable by means of a Finite-State Automaton (FSA).

5.6 Re-Training Actualization

The re-training is the ultimate purpose of the Beholder. When needed, the latter sends a MQTT message to Agorà. From that moment it is Agorà which, with new functionalities we implemented, carries out all the processing required to actually prepare for a new training phase. This requires that:

1. The model, now invalidated, is canceled from the storage. Since the model is saved in a table, we drop the entire table;
2. The DoE, prepared the first time Agorà trained the model for the application and saved as a table in the storage, is restored to its initial stage. This is because every configuration to be explored is associated with a counter, which represents the number of times such configuration still has to be explored by the nodes. At the end of a training phase all these counters are 0. When we need to perform a new training they must be reset to their initial value. In order to know which was the

original value we save it in a new, untouched, attribute of the DoE table. Then, at every retraining request, the actual counter gets reset to this original value;

3. The trace table is truncated. Meaning that every entry of such a table is deleted. There is also the Beholder program option "`no_trace_drop`" which, instead of truncating the table, just deletes its content from the top (oldest records) to the detected change (included), thus keeping the entries of the population after the change;

4. The configurations to be explored are sent to the available nodes. This resets the application model from the clients' knowledge, thus stopping them from sending observations to the Beholder, until a new model is received.

5.7 Class Diagram

Figure 5.11 shows a streamlined version of the Beholder class diagram, in particular, it focuses on the core functions and on those elements we mentioned thus far. Not included is the main method and entry point for the Beholder process; it manages the program options, takes care of setting up the communication with the Cassandra DBMS and with the Eclipse Paho MQTT broker. It also subscribes to the MQTT topics to receive the different LWT, the welcome message and the broadcasted models from Agorà and the observations from the clients. Eventually it setups the thread pool and makes every thread execute a worker function, which does nothing but listening to incoming messages. Once a message is received, the thread which "caught" it in the first place parses it and takes care of its processing.

It is worth mentioning that, as explained before, the logic which "follows" each application's behavior is inside the "`RemoteApplicationHandler`" object. Thus a thread, to carry out the required task, needs to use the handler in charge of the application sender of the message in analysis. In order to do so, we built a dispatcher entity, the "`GlobalView`", which is an intermediary between the working thread and the handler. Essentially, once it is provided with the name of the application, it returns the correct handler on which the thread has to work. This is what the public method "`get_handler`" does. This dispatcher works through a straightforward hash-map ("`handled_applications`") which binds the name (string) of the application to the pointer to the corresponding handler.

We just reported the main public methods and core members. Here is a short explanation of the methods listed in the "`RemoteApplicationHandler`" class:

- `set_handler_ready` sets the current `status` to `READY`. It it used once a new broadcasted model sent by Agorà is received, after a previous re-training request set the handler in the `TRAINING` state;

- `pause_handler` is triggered when the Agorà LWT is received. It involves the saving of the current `status` in the `previous_status` variable, and the setting of the `status` to `DISABLED`;

- `un_pause_handler` is run once Agorà comes back online, after a previous disconnection. The Beholder is aware of such event because it receives the Welcome message from Agorà. It restores the value contained in the `previous_status` in the `status` variable;

- `bye_handler` removes the client who sent the message from the list of active clients. It is triggered when the Beholder receive a LWT from a client;
- `new_observation` is the most complex and frequently used method. As explained before, it involves the parsing of the observation from the client, then if the latter is not blacklisted (and the handler is `READY`) the residuals are computed and put in the corresponding buffers. Next, if at least one residual buffer is full, the first level test (ICI CDT) is performed. In the event that an anomaly in the expected behavior is detected, the second level test (hypothesis test) is carried out too. These two tests are intentionally implemented in separated classes, to better differentiate the logic among the different tasks.

While the worth-mentioning attributes are:

- `mutex` is the object on which the lock is applied, to avoid concurrent unwanted access to the critical data structures;
- `status` represents the current state of the handler, as explained in Section 5.5. In particular, the range of values which this variable can assume is listed in the `ApplicationStatus` enumeration (actually it cannot be set to `RETRAINING`);
- `previous_status` works as the `status` variable, with the only difference that it cannot be set to `DISABLED`;
- `client_blacklist` is a set keeping track of the blacklisted clients for the current run of the first level test, as shown in Section 4.4;
- `clients_list` is the list of active clients running the application associated with the handler in question; it is built as an hash-map whose key is the client name, while the associated value is the timestamp, implemented as a struct, representing the moment the client has been seen, by the Beholder, for the first time, i.e. the time of the first message received from it. Such time is used to delimit the data retrieved from the trace in the second level test, as explained in Figure 5.7;
- `residuals_map` is the hash-map which contains the buffer of residuals (value) associated with each metric (key). The buffer itself is a vector of size ν, i.e. the length of the subsequences on which the ICI CDT works (Section 4.3). Again every residual, which is the absolute difference between the observed value and its estimate from the model, is implemented with a struct, whose components are the actual residual and the timestamp associated with it. The information about the time is needed later on for the second level test to pinpoint, in the trace, the window of time in which the change was detected.

5.8 Conclusions

In this chapter, we focused on some implementation details of the Beholder module. In particular, we illustrated that it is a process placed on a remote server. We described the

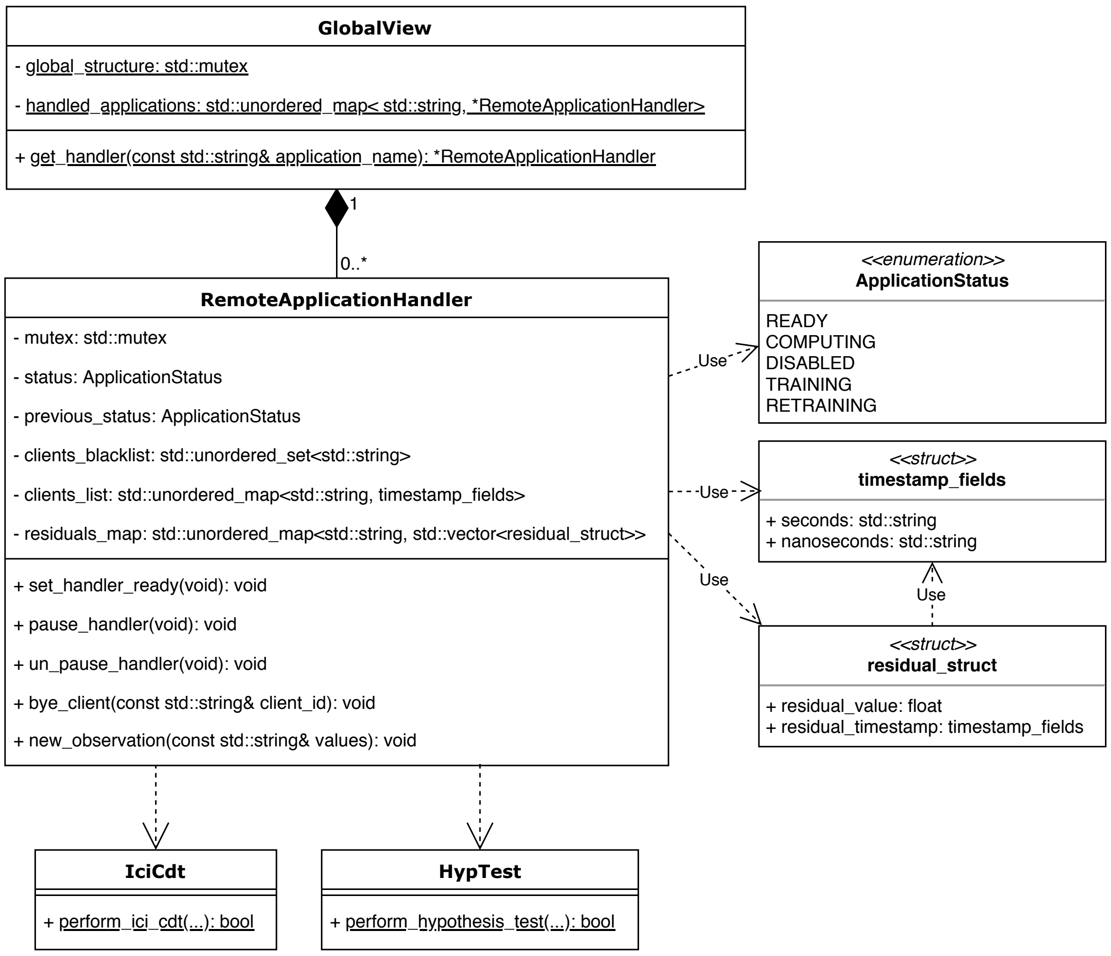

GlobalView
- global_structure: std::mutex
- handled_applications: std::unordered_map< std::string, *RemoteApplicationHandler>
+ get_handler(const std::string& application_name): *RemoteApplicationHandler
1
0..*
RemoteApplicationHandler
- mutex: std::mutex
- status: ApplicationStatus
- previous_status: ApplicationStatus
- clients_blacklist: std::unordered_set<std::string>
- clients_list: std::unordered_map<std::string, timestamp_fields>
- residuals_map: std::unordered_map<std::string, std::vector<residual_struct>>
+ set_handler_ready(void): void
+ pause_handler(void): void
+ un_pause_handler(void): void
+ bye_client(const std::string& client_id): void
+ new_observation(const std::string& values): void
Use
<<enumeration>>
ApplicationStatus
READY
COMPUTING
DISABLED
TRAINING
RETRAINING
Use
<<struct>>
timestamp_fields
+ seconds: std::string
+ nanoseconds: std::string
Use
Use
<<struct>>
residual_struct
+ residual_value: float
+ residual_timestamp: timestamp_fields
IciCdt
+ perform_ici_cdt(...): bool
HypTest
+ perform_hypothesis_test(...): bool

Figure 5.11: *Partial class diagram showing the most important components underlying the Beholder implementation.*

initial knowledge sharing with Agorà, that provides the Beholder with an overview of the whole situation of the framework. We explained how, once online, the Beholder is kept updated on the evolving context of the framework. We outlined all the interactions with the other components of the framework, namely the client nodes, Agorà and the storage. We provided an overview of the main parameters which tune the sensitivity of the two levels of the hierarchical CDT. We explained the functioning of the thread pool structure and the thread safety policies implemented. We illustrated the stateful nature of the Remote Application Handlers together with the characterization of their states. Finally, we described how the re-training process is actually carried out. In the next chapter, we will present the experimental results to validate the proposed approach at the problem of anomaly detection in the behavior of the monitored application and consequent knowledge update.

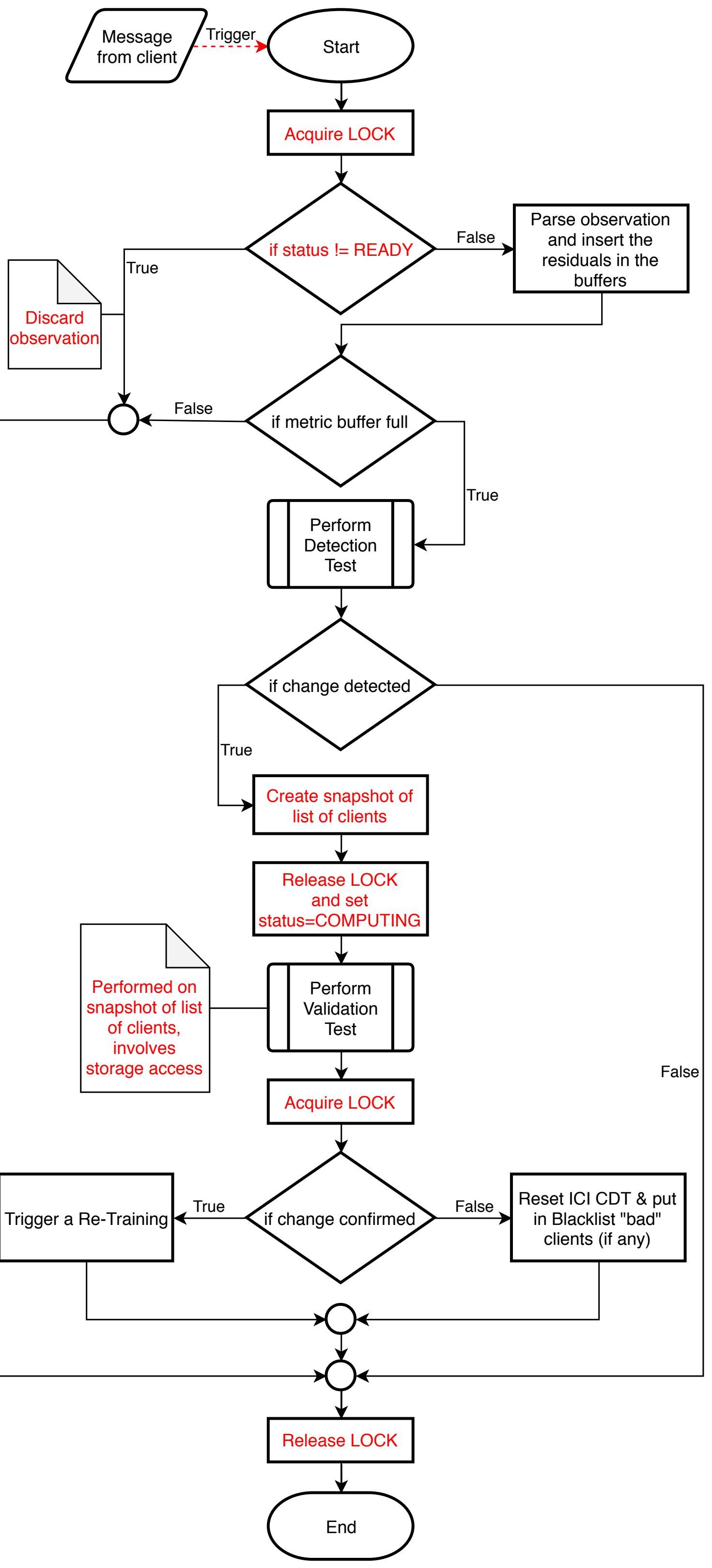


Figure 5.12: *Flowchart presenting an high level overview of the hierarchical CDT in the* `new_observation` *method. Highlighted is the lock/release of the mutex and the use of the handler's status to control the application, plus the snapshot copy of the clients list before entering the second level test.*

CHAPTER 6

Experimental Results

In this chapter we present the experimental results to assess the proposed methodology and to validate the effectiveness of the Beholder module and its contribution in the mARGOt framework. First, we treat the matter of choosing representative and good-enough Beholder parameters for the most universal use cases. We will focus on the first level test, and in particular on the problem of finding the trade-off between the delay of the anomaly detection and the false positives rate. In order to do so we will employ a synthetic application.

Then, we will present as benchmarks for our tests two real-world case studies. In particular, the K-means clustering application will be used to evaluate the reaction mechanism at local level, actuated by the mARGOt autotuner, and at global level, actuated by the Beholder module. The second benchmark is carried out using the Stereomatch application; in this case we, will concentrate on the second level test, evaluating the robustness of the approach when we relax the i.i.d. assumption. We will highlight the strong dependency of the collected data from such assumption and the importance of the stability of the machine on which the application is being run.

6.1 ICI CDT Parameters Analysis

The Beholder module exposes many functional parameters which allow the tuning of the two-level hierarchical CDT. These parameters mainly impact on the performance of the CDT itself, i.e. the Recognition Delay (RD), also known as Detection Delay (DD), and on the ratio of False Positive (FP) and False Negative (FN). The ultimate goal of this analysis is to provide the user with parameters which should be a good fit for the majority of the scenarios. Obviously, should the suggested default options not meet the expectations and requirements, we encourage a more fine-grained application-specific tuning of the parameters.

Let us reflect upon the fact that, in our context, where efficiency is often the main goal, a false positive, that is a detected change when there is not, is often way worse than a false negative. This is because the re-training is an “onerous” decision since it is a time-consuming task; thus issuing such a request when it is not necessary becomes a heavy burden. We can tolerate a detection delay in return for a more confident decision; this is also possible thanks to the corrective feedback integrated in the autotuner which, in the meantime, attempts to ensure the computation correctness despite a possibly erroneous model. Thus we should provide default parameters which represent a good compromise between the delay in the change detection and the FPs rate; ideally, these should be flexible enough not to keep detecting continuous non-existing changes (False Positive), even if the second layer prevents the actual re-training.

The algorithm underlying the ICI CDT (Subsection 2.3.5) is prone to detect a change in any case, sooner or later; it is a structural drawback intrinsic in its methodology. The whole mechanism of the Intersection of Confidence Intervals is based on an ever-narrowing interval which tends toward the real mean of the monitored process; when the interval is small enough, every minimum variation triggers the change. We took this into account, and to reject false positives triggered by such situations we rely on the second level test, which should reset the first level to its initial “wider” interval. Clearly, even in presence of the hierarchical mechanism, we want to avoid as much as possible frequent occurrences of such happenings.

We propose an analysis whose objective is the evaluation of the first level test parameters and their impact on the performance of the test itself. We created a synthetic application in which we can introduce a change at any moment. We will try to find the best trade-off that minimizes both the False Positive and Detection Delay, which are contrasting goals deeply application-dependent. The synthetic application aims at reproducing the streaming observations coming from a real application, which could be the target application throughput. It outputs observations in the neighborhood of a chosen value 1, with an additive simulated noise produced by a Gaussian distribution $\sim \mathcal{N}(0,\ 0.05)$, that is to say centered around 0 and with a standard deviation of 5%, which according to the “68-95-99.7” rule should, at most, have spikes of about $3 \times \sigma$, that in our case corresponds to a 15% noise around the mean, thus yielding observations in the range $[0.85, 1.15]$. We generate, at most, 10000 production samples. If, by then, no change has been detected yet, we stop the data generating process anyway. This experiment aims at observing the behavior of the framework in two different scenarios: (i) the time required to trigger a FP when there is no change, and (ii) the time required to detect an enforced change. All of these considerations are carried out in function of the exposed parameters.

Let us remind that, as far as the first level test is concerned, there are three main settings; the number of observation subsequences used for the CDT training phase, the size ν of these windows, and finally Γ, the amplifying factor of the confidence intervals on which the test is based.

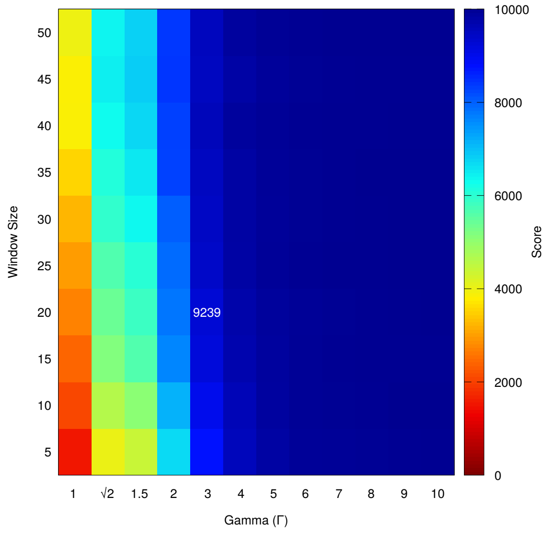
We decided to take into account four different possibilities for the number of training subsequences: 5, 10, 15, 20. We did not test higher values due to the fact that we want to start the model verification process as soon as possible, besides granting the process stationarity required for the training can be tough, thus paradoxically a long training could turn out to be counterproductive if “wrong” behaviors get observed during this learning phase.

As far as the window size is concerned, we tested a range from 5 to 50 sample-per-window, with increments of 5: $\nu \in \{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$.

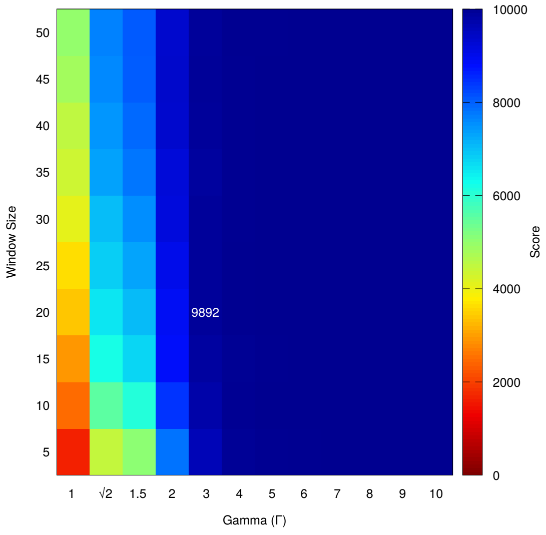
Since we cannot guarantee that the Beholder-monitored metrics have a Gaussian distribution, for the Γ we rely on Chebyshev’s theorem, that provides confidence intervals even for distributions that are not normal. It guarantees that no more than a certain fraction of values can be more than a certain distance, measured in standard deviations, from the mean; in particular: $\Gamma \in \{1, \sqrt{2}, 1.5, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. It states that, the 99% of the samples taken from a distribution, will lie within 10 standard deviations around its mean.

Thus, the objective of the first scenario we analyze is to assess the tendency to trigger a FP change detection at some point. We simulate a constant behavior with our application and compute the rapidity with which the change is detected for every combination of parameters. In particular the four heatmaps in Figure 6.1 are respectively for the different number of training windows considered, i.e. $5, 10, 15, 20$. The score represents the average number of production samples before the detection of the nonexistent change, for every combination of training dataset size, subsequence size and Γ, the CI amplifying factor. The training phase is not reported, and every version of the application is run on, at most, 10000 production samples. If no change has been detected by then, we assign that run the maximum (best) value of 10000. Highlighted is the value for the combination ($\Gamma = 3$, $Window\_Size = 20$), which is the one we designated and that we will use in the tests. The best performers are those parameters for which the -non existing- change is detected as late as possible. Ideally, the top right-hand corner contains, for the considered case, the best combinations of window size and Γ. Thus the deep blue values, tending towards 10000, are the best. Their FP rate is the lowest. On the other hand, the red zones are those which are clearly too sensitive to small acceptable variations due to noise. It is worth mentioning that the numbers hereby reported are results of the averaging of 1000 single runs carried out with different random seeds, for every combination. This is to take into consideration the seed-dependency of the final outcomes, since the random Gaussian noise generator we use is obviously based on a seed. We want to safeguard from particularly “(un)lucky” seeds. From Figure 6.1 one can deduce that the size of the training dataset does not impact excessively after a certain value. For this reason we decided to keep the number of training window quite low, since there is always the strong assumption, for the test to work, that the training has to be carried out in a stationary situation, in which the test “learns” the acceptable range of values for the metrics involved in the analysis. Such stationarity is not easily guaranteed, in fact often the user has no real control over this. Thus we chose to use 10 training subsequences; this will be fixed for the rest of the tests.

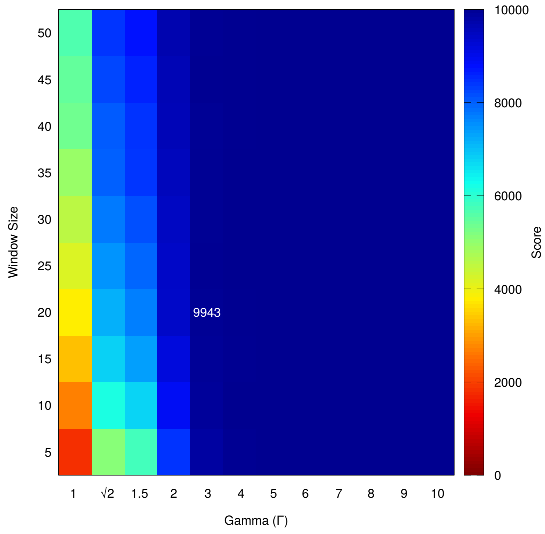
To better characterize the correlation between the test parameters and the corresponding FP rate, we computed the variance of every combination. To be more accurate, we assessed the so-called Coefficient of Variation (CV), also known as Relative Standard Deviation (RSD), which is a standardized measure of dispersion of a probability distribution or frequency distribution, often expressed as a percentage, and is defined as the ratio of the standard deviation σ to the mean μ. This allows having an insight into the statistical behavior with respect to the seed-dependency.



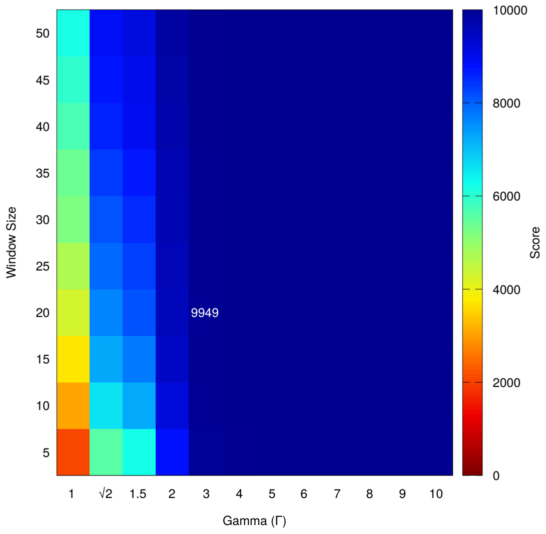
(a) 5 *subsequences used for training*



(b) 10 *subsequences used for training*



(c) 15 *subsequences used for training*



(d) 20 *subsequences used for training*

Figure 6.1: *Outcome of the ICI-based CDT applied on data where a small amount of noise is simulated, but no change takes place. The score represent the average number of production samples before a FP detection. On the x-axis are represented the values for* Γ*, the CI amplifying factor, while on the y-axis the subsequence sizes.*

Figure 6.2 presents the analysis of the RSD for the case (b) depicted in Figure 6.1, that is the version where 10 subsequences have been used for training. The score represents the RSD for every combination of parameters which is being run with 1000 different random seeds. As usual, the deep blue regions are the best from the point of view of the FP rate, and in this case are also the most consistent ones; they are almost seed-independent and represent stable-enough combinations of parameters which can be employed. Highlighted is the value for the combination ($\Gamma = 3$, $Window\_Size = 20$), which is the one we designated and that we will use in the tests. In Figure 6.2 it is evident how unstable the bottom left-hand side zone of parameters combinations really

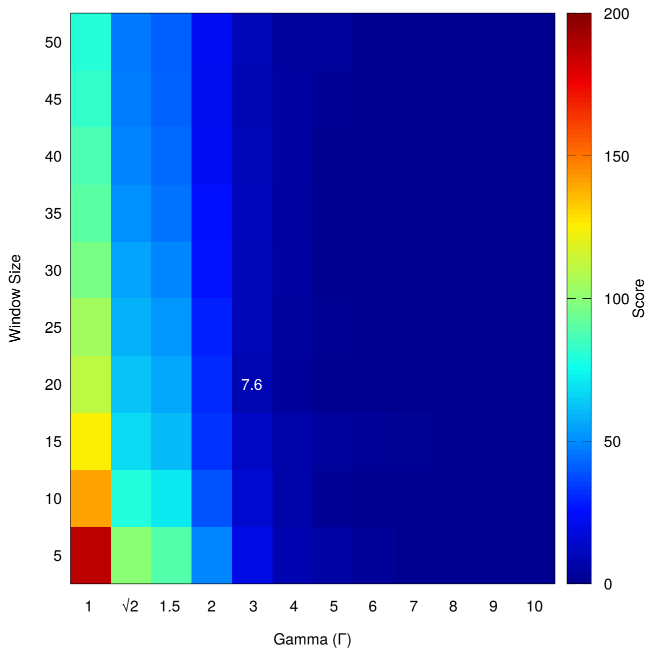
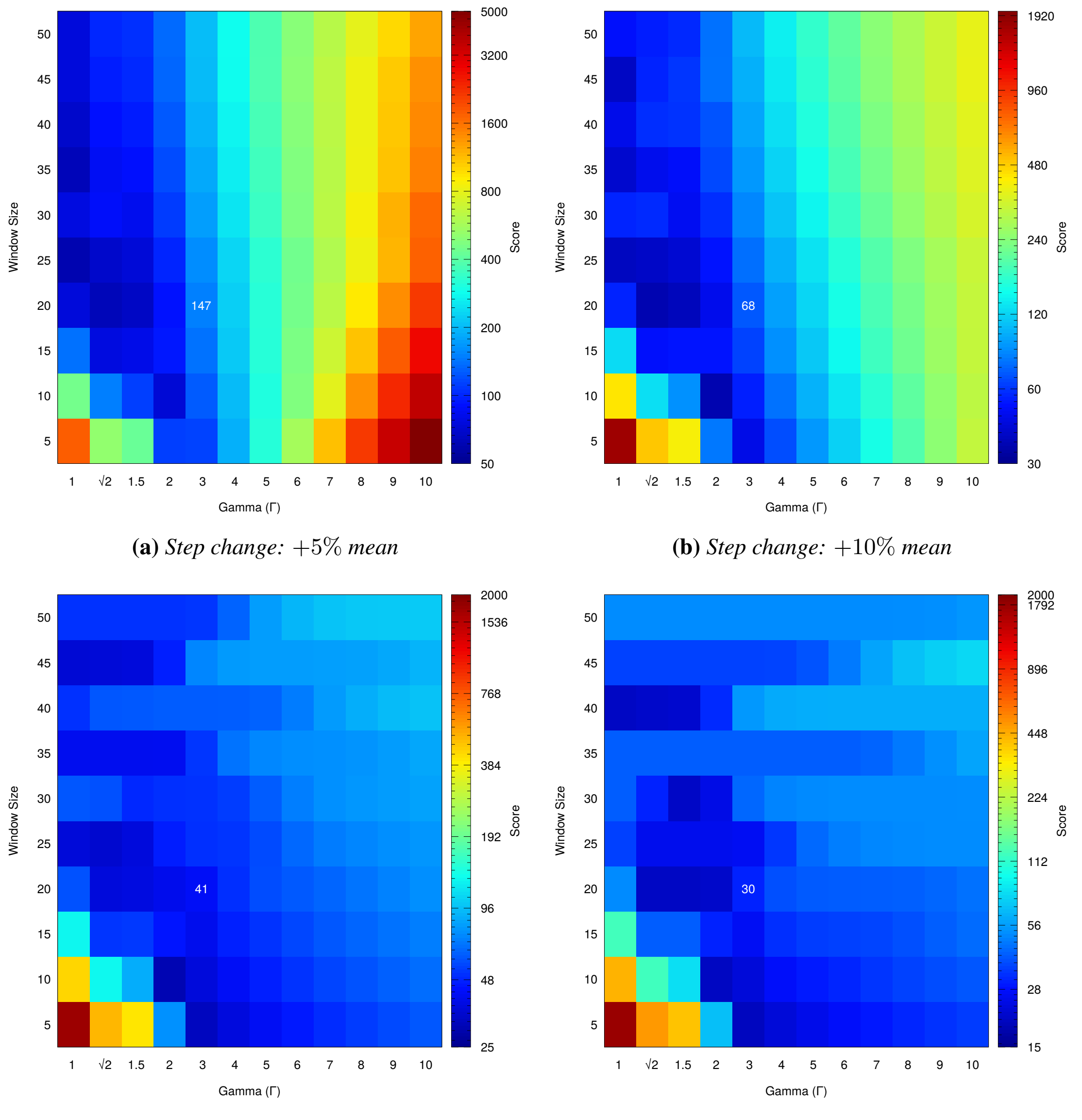


Figure 6.2: *Analysis of the RSD for the case (b) depicted in Figure 6.1. The score represents the RSD for every combination of parameters. On the x-axis are represented the values for* Γ*, the CI amplifying factor, while on the y-axis the subsequence sizes.*

is, where the RSD reaches its maximum peaks. This provides further confirmation of the fact that ideally, the higher both Γ and the windows size parameters are, the better it is, at least from the FP rate standpoint.

Unfortunately, a high value of Γ is in contrast with the DD, thus we need to find a set of parameters which is a good compromise between the FP rate and the DD. This naturally counterbalances (lowering) the values for both the windows size and Γ. Thus, in the second scenario we simulate a change and evaluate the Detection Delay. In particular we decided to simulate two different kinds of changes, the first being a step, where we enforce a sudden increase in the generated data in one case of 5% and in the other of 10%, while in the second version we apply a linear increase in the data, respectively of 1% and 3%. To put this into context, this simulates an anomaly in a monitored streaming process. What we measure is the average number of elapsed samples, out of the usual 1000 runs with different seeds, between the moment we apply the change and the actual detection of such change. Figure 6.3 presents the analysis performed in this second scenario, with the four different enforced changes and the respective recognition delay. In particular, the score represents the Detection Delay, which is the time it takes before the enforced change is acknowledged. In all the cases 10 windows have been used for training. Please note that the score is on a logarithmic scale. The deep blue zones represent the best set of parameters as far as the DD is concerned. On the contrary, the bottom left-hand zone for all the four pictures contains all those configurations for which a nonexistent change was detected before the actual enforced one. This falls back in the first scenario analysis presented above and confirms that these are FPs. We assigned to these wrong parameters the worst value of the range, to underline their corresponding unacceptable performance. Highlighted is the value for



(a) *Step change: +5% mean*

(b) *Step change: +10% mean*

(c) *Linear change: +1% mean*

(d) *Linear change: +3% mean*

Figure 6.3: *Representation of the Detection Delay, which is the time it takes before the enforced change is acknowledged. The score is in a logarithmic scale. On the x-axis are represented the values for* Γ*, the CI amplifying factor, while on the y-axis the subsequence sizes.*

the combination ($\Gamma = 3$, $Window\_Size = 20$), which is the one we designated and that we will use in the tests. It is evident how, in this situation, the top right-hand zone does not represent the best set of parameters as it did in the first scenario in Figure 6.1, where the FP rate was being optimized. This proves the contrasting natures of the two metrics for which we have to find a trade-off.

We designated the combination of parameters ($Training\_Windows = 10$, $\Gamma = 3$, $Window\_Size = 20$) as flexible enough to be considered a good compromise between the FP rate and the Detection Delay that we present below. Obviously, one could argue that being our simulated noise Gaussian, the $\Gamma = 3$ is a straightforward result according to the "68-95-99.7 rule", the so-called "three-sigma" rule of thumb which expresses a conventional heuristic that nearly all values of a normal distribution are taken to lie within three standard deviations of the mean. Nevertheless, we also tried to carry out the same analysis with a uniformly distributed generated noise, without any real appreciable difference in the observed results. Moreover, these values proved to be yielding satisfactory and robust results also in the use cases we tested and that we will propose later on in this chapter. Nonetheless, we stress the fact that the final user should preferably tune these application dependent parameters according to the observed test performance. Bear in mind that we analyze the observations from a streaming process. The filling frequency of the ICI test subsequences depends on the incoming observation rate of the analyzed process. When a window is full the first level test performs its check, thus another metric of tuning of the subsequence size is according to the desired periodicity of such test. Moreover, the detection delay depends also on the magnitude of the anomaly, and on the number of deviating nodes with respect to the total ones.

Please note that this analysis has been carried out just on the sample mean feature, while the ICI-based CDT also offers the sample variance feature (Subsection 2.3.5); since for our usage of this statistical test in the mARGOt framework we rely mainly on the former feature, we wanted to ensure consistency in that regard. We also tested the functioning of the sample variance with the selected parameters, by enforcing a change on the variance of the simulated noise, and even though we are not reporting the results here, it returned coherent results in the detection of the change, with performances (DD and FP) comparable with those measured on the sample mean.

6.2 Case Study: K-means Clustering Application

What we aim at showing with this experiment is the operation of the modified version of the mARGOt framework enhanced with the new Beholder module. We will demonstrate the operation of the two reaction mechanisms at different layers, to cope with a variation in the execution environment that allows the adaptation to the new context, namely the preexisting autotuner MAPE feedback loop and the newly introduced Beholder. In this scenario, we propose we use the K-means clustering application.

K-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining; it aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. This results in a partitioning of the data space into Voronoi cells, that is a division of a plane into regions, based on distance to points in a specific subset of the plane. The problem is computationally difficult; however, efficient heuristic algorithms converge quickly to a local optimum. The most common algorithm uses an iterative refinement technique. Due to its ubiquity, it is often called the k-means algorithm; it is also referred to as Lloyd's algorithm, particularly in the computer science community. Given an initial set of k means, the algorithm proceeds by alternating between two steps, as shown in Figure 6.4:

1. Assignment step: Assign each observation to the cluster whose mean has the least squared Euclidean distance, this is intuitively the "nearest" mean;

2. Update step: Calculate the new means (centroids) of the observations in the new clusters.

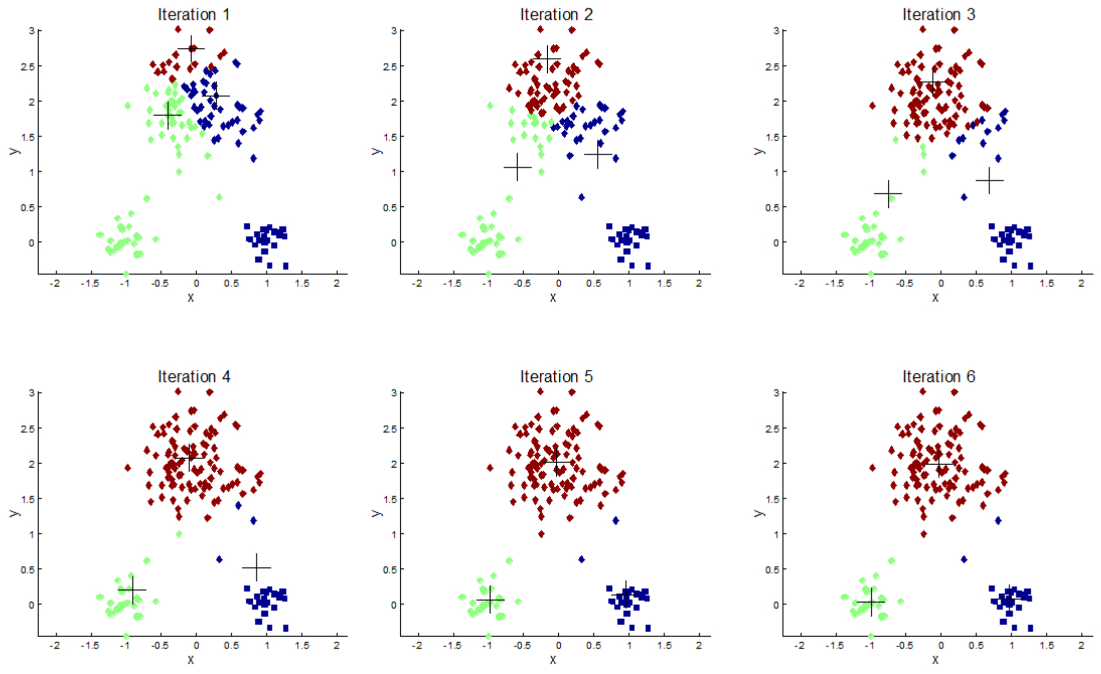


Figure 6.4: *K-means algorithm. Observations are shown as dots, and cluster centroids as $+$. In each iteration, we assign each observation to the closest cluster centroid (shown by "painting" the observations the same color as the cluster they are assigned to); then we move each cluster centroid to the mean of the points assigned to it.*

The algorithm has converged when the assignments no longer change; this approach does not guarantee convergence to the global optimum and the result may depend on the initial clusters. The complexity of the algorithm depends on the input data, namely on the number of rows (n) and attributes (every observation is a d-dimensional vector) in the dataset.

Our application is tunable in the maximum number of consecutive iterations. In our case, such knob can assume any value in $\{1, 2, 5, 10, 20, 50\}$, where 50 is what we consider the optimal reference against which we compute the approximation error. This kind of application satisfies the requirement of providing i.i.d. observations required for the application of our tool. We will use the Beholder to monitor the correctness of the expected execution time.

Since the execution time varies with the input size, if we do not take the input dataset features into account we break the i.i.d. assumption, because the residuals computed by the Beholder would obviously depend on the size of the input file currently processed, which changing continuously would render meaningless a statistical analysis based on the stability of the monitored process; an anomaly would be detected at every change of input size. Such a situation needs to be dealt with at model-level, so that the autotuner itself can correctly be proactive with respect to the input data features. This also takes us back to an i.i.d. situation, because a different configuration will be enforced by the autotuner for every dataset size. We generated a very high number of datasets that the application will process; all of them have the same number of dimensions, but they can vary in the number of observations n, in particular, $n \in \{1000, 4000, 7500, 12500\}$. The order in which they are processed is casual. Thus, we instruct Agorà by specifying that we will provide datasets of four different sizes, and that this feature has to be taken into account, by basically making four different models, one for each size. As fixed constraint we require that the computation of a single dataset does not take more than $10000\mu s$; moreover, we require the most accurate result which still satisfies the goal, meaning that we want to minimize the error as much as possible.

Two servers were employed for the experiment. On one of these we run Agorà and the Beholder module, it acts as a server. The other machine runs four instances of the K-means application, thus representing the client(s). We used two devices simultaneously to both test a real world client-server use of the proposed tool across a network, and to avoid bias on the measured execution time that could be introduced when running both the server and client on the same computer. The worker machine is equipped with a quad-core processor, precisely an Intel® Core™I7-2630QM, and 16 GB of ram.

Before running the experiment we disabled the CPU Turbo Boost technology to create a stable environment and not to break the i.i.d. assumption required for the correct functioning of the whole CDT process. This should prevent any automatic dynamic frequency scaling. Moreover, we set the clock frequency of the processor to 2GHz, which is not the maximum for the employed CPU. This is to further avoid any possible thermal throttling introduced automatically to safeguard the hardware from dangerous overheating that could override our fixed frequency. The experiment is run in such stationary condition; the training of the application model and the CDT itself are carried out in this context. Then an anomaly is introduced by suddenly lowering, intentionally, the CPU frequency on all the cores. This is put into practice by changing the CPU governor from "performance" to "powersave" using the "cpupower" tool. This aims at simulating real-world scenarios in which, usually for heat-related issues, a power throttling takes place. The algorithm performance is linear with respect to the input size and to the CPU frequency. Being the correlation linear, we show how also the autotuner is able, in such situations, to successfully adapt to the new context thanks to its MAPE loop.

Figure 6.5 shows the behavior of the local autotuner on one of the four clients. In the current experiment, all the nodes behave approximately the same, thus we present just one of them. It is evident how, at first, since the framework does not have any previous knowledge on the application, Agorà takes care of the learning phase. During this profiling stage, highlighted with a yellow background in the figure, the goal is not satisfied, because the application model is yet to be computed. Once the model is available it is clear how the autotuner is able to correctly satisfy the requirements on the execution time. The correctness of the application model employed by the autotuner is proved by the overlapping plots of the predicted performance retrieved by the model and the observed one gathered by the time monitor. The noisy nature of the curves is due to the fact that the input datasets are continuously changing, so different configurations are applied at any moment. When we manually change the CPU governor it is evident how, in the beginning, the goal is not satisfied. It takes time for the MAPE loop and the overall autotuner reaction mechanism to converge to the linear correction. This corresponds to the area in the figure with a purple background. Since the execution time response is linear with respect to the CPU frequency, the autotuner is able to correctly cope with the change. In the meantime the Beholder, which was carrying out its analysis in parallel, detects the irregularity too. We will show later on its functioning. Once both the two

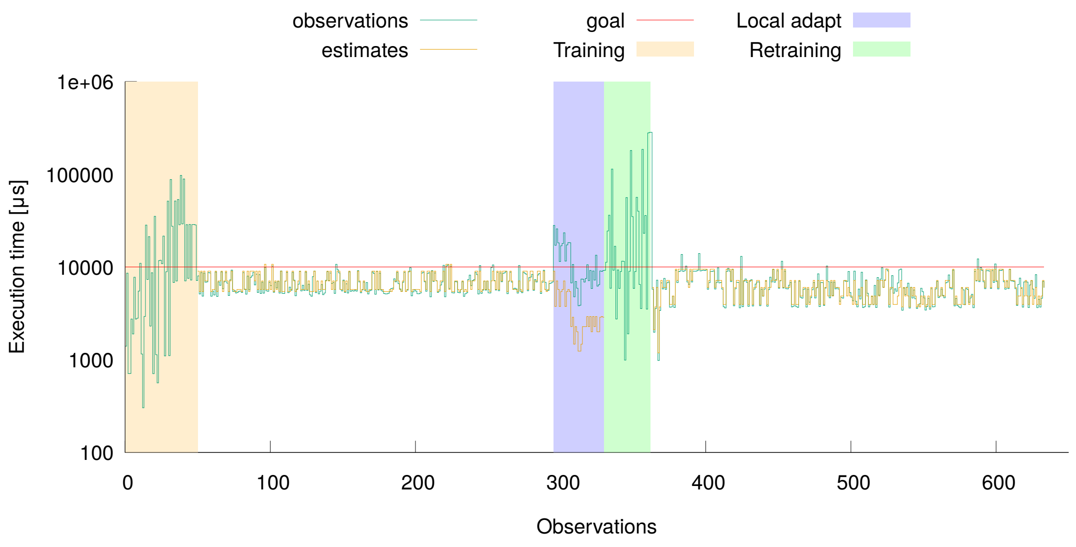


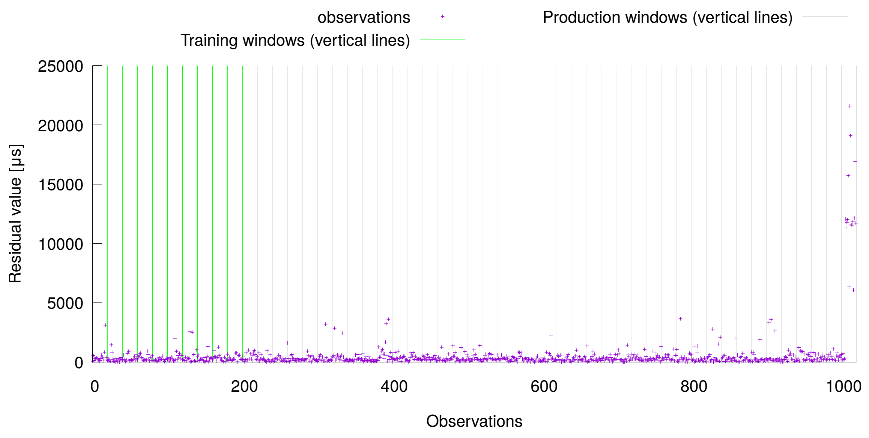
Figure 6.5: *Plot of the observed and expected execution time metric collected from one of the running nodes. Highlighted with different colors are the various stages of the process and the tuning itself performed by the local autotuner.*

levels of the hierarchical CDT confirm the change, a re-training request is issued by the Beholder. From this moment, the application model is deleted and the process restarts from scratch. Agorà carries out a new training phase, pictured with the green background in the figure. Once the training phase finishes and the model is delivered to the clients, these start again running the application with the autotuner-provided parameters.

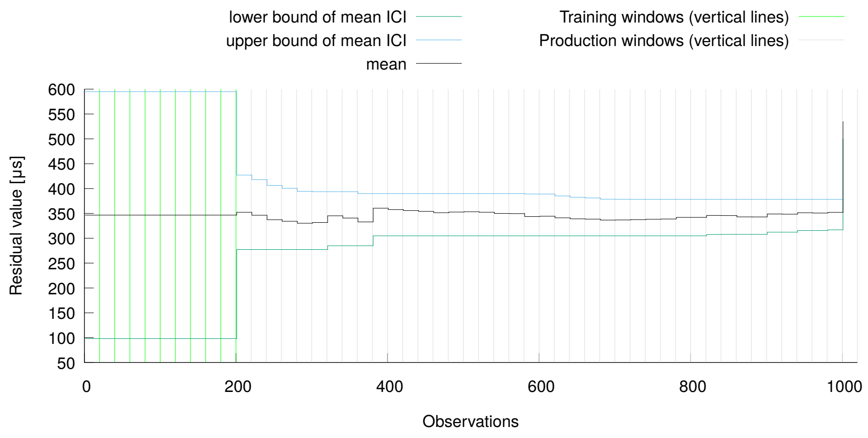
Now we will analyze this same process from the Beholder viewpoint, presented in Figure 6.6. Please note that the plot in Figure 6.5 is the one produced by the local autotuner on one of the four clients, thus it refers to one node only, while in Figure 6.6 is shown the log of the first level CDT carried out by the Beholder module on the server; it represents the collective behavior observed from all the four clients together. In this analysis, we used 10 windows for the first level test training, where every window contains 20 observations. We used $\Gamma = 3$.

Figure 6.6a clearly shows that after about 1000 observations analyzed, the residuals start to diverge. That is when we changed the CPU governor. Being the residuals the absolute difference between the estimated time (computed with the performance governor) for a given execution and the observed execution time, as soon as we set the powersafe governor we witnessed, as expected, a detection. In particular, Figure 6.6b shows how the change was detected thanks to the sample mean feature; the sample variance feature test was not carried out on the last window (Figure 6.6c), since the test on the sample mean had already triggered the change. In such cases, we pass to the second level test without losing any precious time in the computation of the other feature.

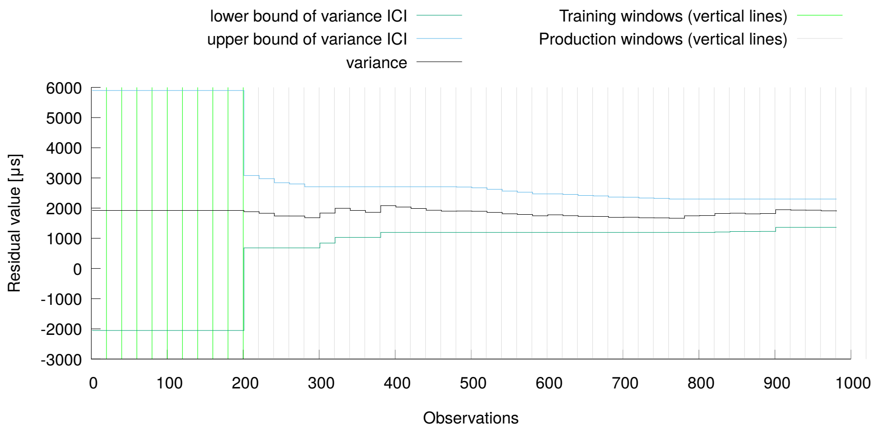
Since the enforced change acts on all the cores of the worker machine, all the four nodes will experience the irregularity. The hypothesis test, which is carried out with a significance level $\alpha = 0.05$, that with the Bonferroni correction applied becomes $\alpha = \frac{0.05}{4} = 0.0125$, given that there are four tests carried out, one for every node, rejects with huge confidence the null hypothesis that the population before and after the change



(a) *Residuals on which the test works.*



(b) *Test on the sample mean feature.*



(c) *Test on the sample variance feature.*

Figure 6.6: *First (ICI-based) level of the hierarchical CDT.*

window have the same mean, thus confirming the change; this results in a retraining request by the Beholder module toward Agorà.

The outcome of this experiment is that, since the execution time is linearly correlated with the processor frequency, in such scenarios the autotuner is able to cope with the change locally. There are many situations in which this does not apply. We also proved the expected functioning of the new module Beholder in the anomaly detection; the retraining issued by the latter allows the computation of a new updated model which works with the powersave governor; thus, if new clients with the same underlying architecture and configuration join the computation, they are sent the new model directly and do not need to locally adapt. This already represents an increase in the proactivity of the framework in the face of anomalies. The change detection performed by the Beholder is also the unique approach to solve situations in which a linear correction of the model by the autotuner is not able to converge to a solution of the problem.

6.3 Case Study: Stereomatch Application

With the Stereomatch case study, we aim at showing the importance of the i.i.d. assumption in the data collected and monitored by the Beholder module. We will show how strict such applicability requirement is, above all for the second level test. We will provide an example in which it might be worth looking at the practical difference of the detected change, and not only on the statistical significance.

Stereo matching, also known as disparity mapping, is an important subclass of computer vision algorithms to compute depth maps. Mars rovers, Google's new self-driving cars, as well as quadcopters, helicopters, and other flying vehicles use Stereo matching, since it is the simplest way to find depth with two rectified images. It takes as input a pair of images from a stereo camera, and it computes a disparity map of the captured scene, as shown in Figure 6.7. The output of this application is required for estimating the depth of the objects in the scene. The algorithm derived by [73] builds adaptive-shape support regions for each pixel of an image, based on color similarity, and then it tries to match them on the other image, computing its disparity value. The algorithm implementation [64] exposes five application-specific knobs to modify the effort spent on building the support regions and on matching them in the second image to trade off the accuracy of the disparity image (the Stereomatching output) and the execution time (and thus the reachable application throughput). The accuracy metric is the disparity error, defined as the average intensity difference per pixel, in percentage, between the computed output and the reference output. The application has been parallelized by using OpenMP, making available as sixth software-knob the number of threads used for the computation.

We employed an excerpt of the "Tsukuba" dataset [17] composed of 200 pictures. It has been generated using photo-realistic computer graphics techniques and modeled after the original "head and lamp" stereo scene released by University of Tsukuba in 1997. Since we are working on a completely CG generated world, the dataset integrates very accurate depth maps, that we will use as the reference to compare the accuracy of the different configurations.

The application requirements impose constraints on a minimum level of throughput, while minimizing the computation error. In this scenario, the Beholder is interested in changes in the throughput, rather than changes on the error. This is due to the complexity of monitoring the computation error, which is always disabled in the production phase. The throughput is measured as frame rate (expressed in Frames Per Second (FPS)).

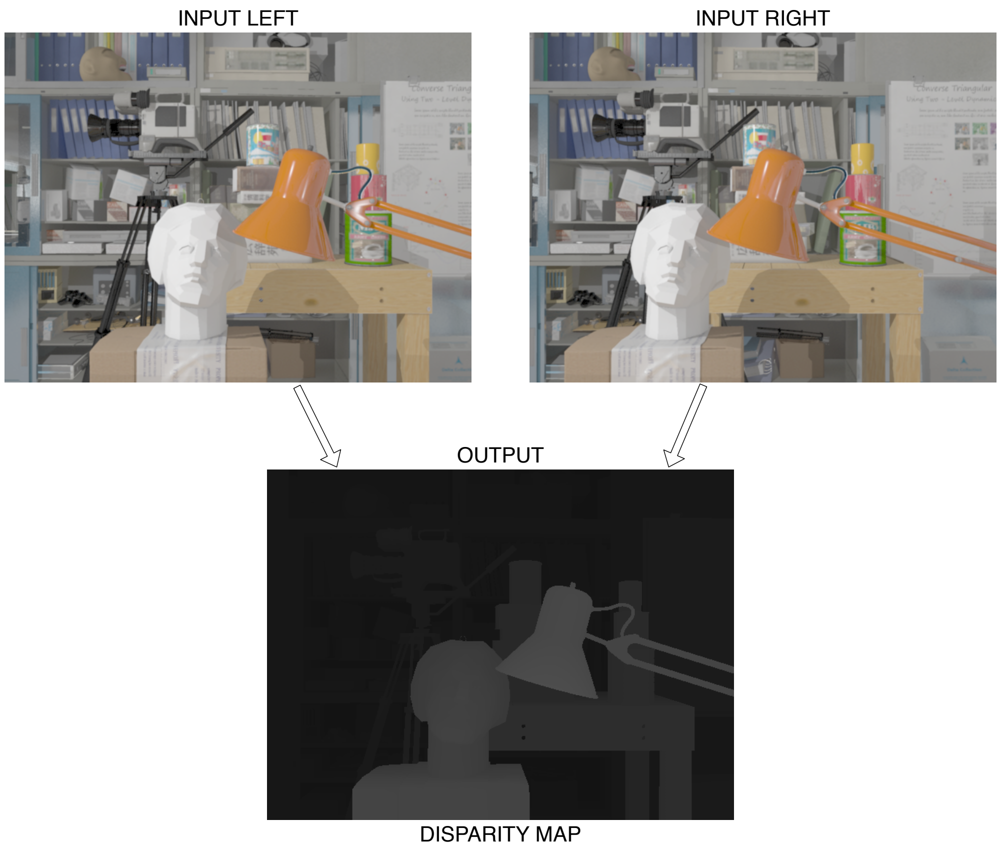


Figure 6.7: *Example of Stereomatch application run on the Tsukuba dataset. The two images above are fed in input to the algorithm, which computes and outputs the disparity map.*

With our module, we will measure the stability of the throughput to spot changes in the application behavior.

It is important to notice that, obviously, the complexity of the algorithm depends on the resolution of the input picture, but also on the content of the picture itself. Thus, processing different pictures already breaks the i.i.d. assumption, since some pictures are more complex than others; thus the measured fps will depend on the picture currently being processed.

The other source of issues that can break the i.i.d. assumption is represented by the computer on which the experiments are run. In particular, we witnessed how the dynamic CPU frequency scaling can introduce dramatic time-dependencies in the collected observations; this would violate the underlying assumptions at the base of our work. Thus, in order to work on an as stable as possible architecture, we disabled the Intel Turbo Boost technology, and we fixed the CPU frequency at its maximum by setting the “performance” governor; we used the “cpupower” tool to set such configuration. We run 16 instances of the Stereomatch application, to simulate clients, on a dual socket computer equipped with 128 GB of ram which employs two eight-core Intel® Xeon® Processor E5-2630 v3. Agorà and the Beholder module lie on another machine, which acts as a server.

We will present different versions of the experiment, where we gradually break the i.i.d. assumption, to prove the statistical relevance of such assumption in our approach.

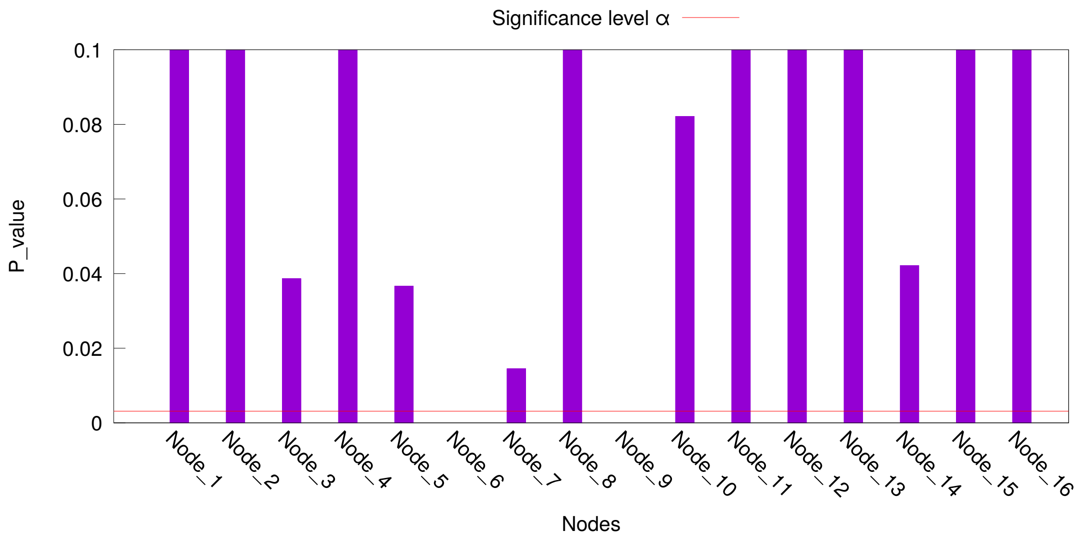


Figure 6.8: *Histogram showing the result of the hypothesis test in the scenario in which two clients experience a change in the input data. The range of the y-axis has been limited to improve the readability.*

6.3.1 Processing of the same frame

In this experiment, we aim at proving the correct functioning of the second level test when the i.i.d. assumption is satisfied, both as far as the application and the underlying architecture are concerned. In order to do so, we cross out the input dependency problem by continuously working on the same pair of input pictures. We keep computing the same disparity map. In this experiment, since there are many running clients sending observation, we increase the size of the subsequences on which the first level works to 40 samples. We fill-in a window in about 1 second, so the check on the observed application behavior is very frequent. We simulate two scenarios, in the first we introduce a change on two clients, in the second on eight clients. What we expect is the change to be detected in both the two scenarios by the first level test. With the difference that, the percentage of misbehaving clients is respectively 12.5% and 50%; this information is provided by the second level test, which works client-wise. We set a maximum ratio of clients deviating from the model of 20%. Then, in the first scenario we expect a rejection of the overall collective change that will reset the first level test and blacklist the two misbehaving clients. In the second case, instead, the change has been confirmed; this will trigger a re-learning of the knowledge, since clearly the current model is not representative of the situation anymore. The change is introduced by making the designated clients run on the same picture with a higher resolution; this obviously lowers the observed FPS with respect to the expected value, thus increasing the residuals computed by the Beholder. The experimental results proved that the first level test works as expected in promptly triggering the detection of the change. The delay in the detection depends on the magnitude of the enforced change and on the number of clients deviating.

Figure 6.8 illustrates the outcome of the hypothesis test in the scenario where only two clients change. The test rejects the null hypothesis when the p-value is below the significance level. According to the Bonferroni correction, the significance level of 0.05

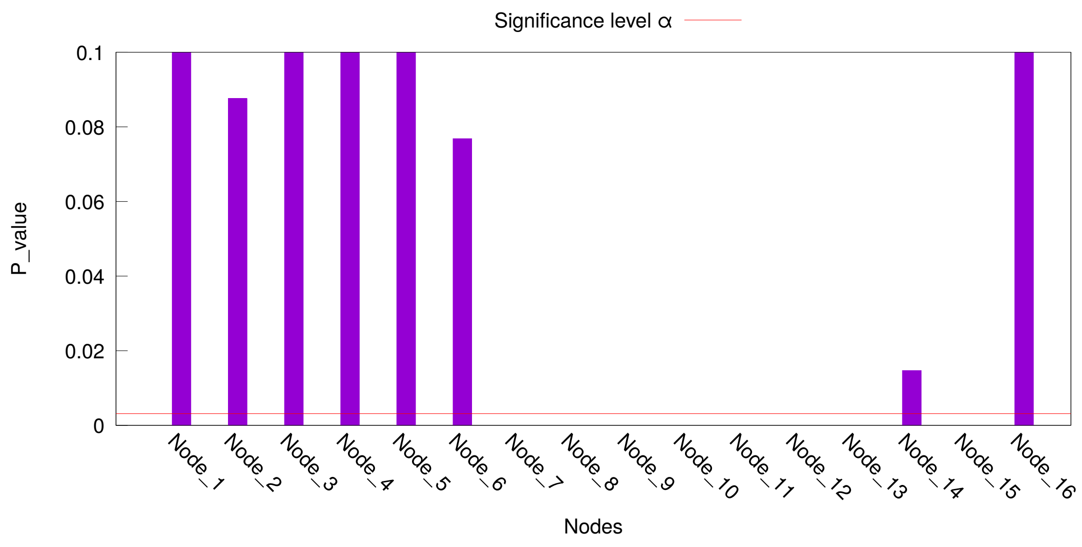


Figure 6.9: *Histogram showing the result of the hypothesis test in the scenario in which eight clients experience a change in the input data. The range of the y-axis has been limited to improve the readability.*

was divided by 16, the number of parallel tests, which corresponds to the number of clients. According to the distribution of the observations for each client, the p-value varies. This is because the process retains its stochastic nature. The worker computer is subject to the underlying operating system noise. Nevertheless, the null hypothesis of equal mean before and after the detected change is rejected for two nodes, as expected; these are “Node_6” and “Node_9”, which will be blacklisted and not taken into account for the next first level test. The ICI test will be reset to its original training phase and the overall change is rejected, given that the misbehaving clients are below the allowed threshold. Figure 6.9 illustrates the outcome of the hypothesis test in the scenario where eight clients change. The considerations on the significance level are the same as the previous scenario. The only difference is in the number of misbehaving clients, which is above the allowed threshold. The second level test confirms the overall change and the re-training request is issued.

6.3.2 Processing the dataset

The performance of the Stereomatch application depends on the input data. In such situations, to maintain the i.i.d. assumption required by our approach, the features of the input data must be taken care of at model-level. Nevertheless, Stereomatch belongs to the class of applications for which the extraction of the input features is very complex. It is not just the resolution of the frames, which can be easily extracted; there is also dependency from the content of the picture. The feature extraction process would introduce a complex overhead, thus it is not performed. This means that the model is not proactive with respect to the input data. We wanted to investigate the effect of the input data variation on the performance of our CDT analysis. Thus we run two experiments; in a first scenario we process the entire dataset sequentially as-it-is, in the second version we randomize the input picture, meaning that we take in input an always different random pair from the dataset. What we want to gauge is the ability of our methodology to cope with the slight variation of the continuously changing data. As in the experiment before, we launch 16 instances of the application. These will process the whole dataset. After a while, we introduce a change in the resolution of the processed dataset on two clients. We expect the framework to reject the overall change and to blacklist the two misbehaving clients.

From the experimental results, we saw how the first level test proved to be robust in any case, at the cost of setting Γ to 4, thus relaxing the rigidity of the test to accept more noise in the observed process. After multiple runs of the experiment, we can say that also the second level test proved to be robust enough, except for some very rare FP outlier in the sequential dataset scenario. With the randomized input, the hypothesis test returns more confident results. This is because there are dependencies on the complexity of the pictures, which vary with time. The frames that compose the dataset are originally from a video where a stereo camera is animated and moved through a computer-generated scene. Thus, when processing the dataset sequentially, adjacent pictures have similar complexity. This breaks the i.i.d. assumption. When considering the randomized dataset, we are able to restore a more stationary behavior, since the complexity of the analyzed pictures varies in a more uniform way during the whole execution. This provides more robust results also in the hypothesis test. This experiment proved that the proposed approach is robust with respect to slight variations in the input data for this application, which are on the boundary of breaking the required i.i.d. assumption.

6.3.3 Breaking the i.i.d. assumption

In this last experiment, we wanted to completely abandon the i.i.d. assumption required by the proposed methodology. We re-introduced the dynamic CPU frequency scaling, i.e. we re-enabled the Intel Turbo Boost technology and we did not fix the processor frequency. We tested with the described setup both the scenarios of the fixed picture and the real dataset.

We repeated the experiment ten times. Figure 6.10 shows an example from one of these runs, in the case in which we enforce a change on just two clients. It represents the outcome of the hypothesis test on a logarithmic scale. As usual, the Bonferroni correction for the significance level was employed. From the results, we can notice how the second level test becomes unreliable once the i.i.d. assumption is not verified. In particular, out of 16 clients, we would expect a situation like the one in Figure 6.8, where 14 nodes are above the threshold. In reality, Figure 6.10 shows how only 8 clients are correctly accepted as nodes whose distribution does not change during the analysis.

As expected, the continuous variation of the CPU frequency introduces dramatic time dependencies which are terrible for our approach. The main source of concern is in the second level test, which loses its role of discriminating layer to reject false positives triggered by the first layer. On the contrary, it often confirms the anomaly. From a statistical viewpoint, this is correct and expected. The variation in the performance of the underlying system yields data whose distribution keeps changing. In order to still provide meaningful information to the user and to realize our objective, which is detecting anomalies in the monitored process, we inserted a third control layer. When the proposed tool is used outside of its underlying i.i.d. assumptions, we propose to validate the change by taking into account what is known as “effect size”. This translates

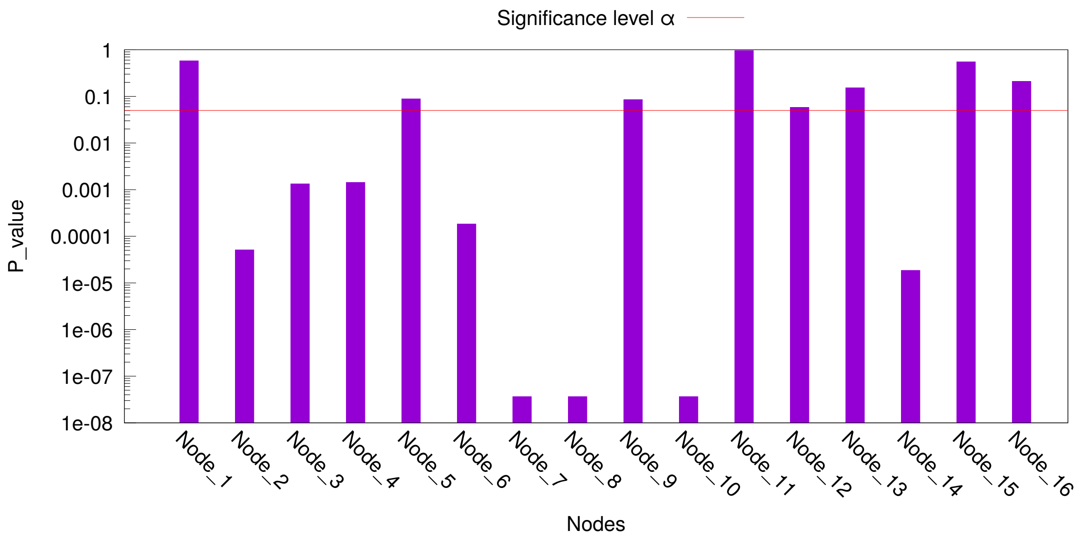


Figure 6.10: *Histogram showing the result of the hypothesis test in the scenario in which two clients experience a change in the input data; case in which the dynamic CPU frequency scaling breaks the required i.i.d. assumption. The range of the y-axis is on a logarithmic scale.*

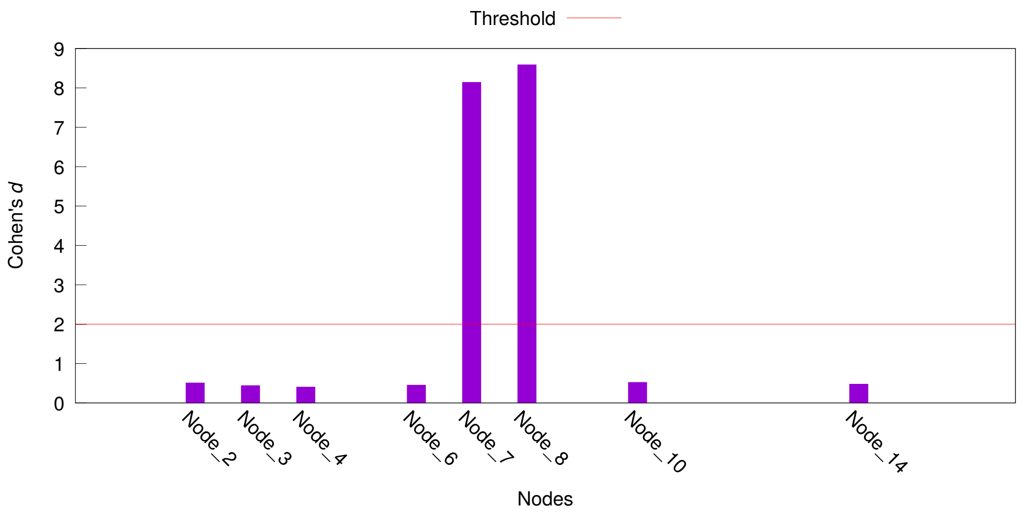


Figure 6.11: *Histogram showing the result of the Cohen's* d *test for the nodes rejected by the hypothesis test in Figure 6.10.*

in evaluating the magnitude of the difference between the two distributions compared with the hypothesis test. When the hypothesis test confirms the change, we offer the possibility to use Cohen's d test to validate such a decision. The idea is to understand whether, in addition to the statistical information offered by the hypothesis test, there is a corresponding meaningful practical difference in the two populations. Relying on the Table 4.1, we set a threshold for the Cohen's d at 2. Even though such an approach implies relaxing the statistical robustness of the final decision in favor of a more empirical evaluation, the observed performance proved to be really consistent with the expected behavior of the tool. Returning to the example in Figure 6.10, for the 8

clients which are rejected, we computed the Cohen's d to evaluate the effect size of the variation in the distribution perceived by the hypothesis test. Figure 6.11 shows the Cohen's d for these nodes. Clearly, only two nodes (Node_7 and Node_8) have a practical difference such as to be considered nodes deviating from the model of a significant margin. These are the two clients on which we enforced the change. The remaining six nodes are way below the threshold, and are the FPs resulting from the execution on CPU cores where the frequency variation took place.

After multiple experiments, we can say that clients on which no resolution change was enforced, when confirmed as misbehaving by the hypothesis test due to the frequency scaling, were always rejected by the third layer. The Cohen's d test makes the framework still able to provide meaningful results when abandoning the i.i.d. assumption.

We conclude by emphasizing the importance of the i.i.d. assumption for the first two level tests which rely on strong statistical significance; that is a requirement for both the application and the underlying architecture. As far as the architectural features are concerned, it is often a common best practice to disable dynamic frequency scaling, Turbo Boost, hyperthreading and similar technologies in HPC facilities; for example, this policy is enforced at Cineca, the largest computing center in Italy. Many of these features were designed, in fact, with the consumer computer use case in mind; actually, for typical desktop applications, the burst in performance provided by the dynamic processor frequency scaling is beneficial. However, we were able to offer a further lifesaver layer to be used in those situations where the i.i.d. assumption is not guaranteed.

6.4 Conclusions

In this chapter, we provided the results of experiments performed to validate the proposed methodology. First, we carried out an analysis to evaluate the effects of the parameters of the first level test on the FPs rate and on the Detection Delay. This was done to help the developer in choosing some good starting point parameters, that otherwise must be characterized for the specific target application. Then, we proposed the K-means case study, that we used to show both the reactive policies of the preexisting autotuner and of the new module, Beholder. Finally, the Stereomatch application was used to better characterize the performance of the second level test with respect to the required i.i.d. assumption. We intentionally broke such an assumption to present evidence that, in the worst case, the third layer constituted by the effect size evaluation can save the situation. In the next chapter, we summarize the contribution of this thesis and we provide ideas for possible future investigations in similar directions.

CHAPTER 7

Conclusions and Future Works

The challenge confronted in this work was trying to relax the assumptions underlying the application knowledge employed by an autotuner, which is treated as ground truth. We presented real-world examples of anomalies introduced from the input data or the execution context viewpoint which clearly invalidate the correctness of a previously computed application model.

We tackled the problem by empowering a preexisting autotuner framework with the ability to detect such context switches, and, if deemed necessary, to update its own knowledge about the application in analysis. In particular, we added the Beholder, an optional module in charge of performing a two-level hierarchical Change Detection Test (CDT), where the first layer carries out the anomaly detection task by evaluating collectively incoming residuals from all the nodes. The second level instead validates the detected change, by enforcing a user-set policy which compares the weight of the misbehaving nodes with respect to all the available. We made the framework compatible with the possible re-training request issued by the new module, automating all the actions required to reset the old model and to compute and deploy the new one to all the working nodes. This required the implementation of a new type of monitor to automatically manage the computation of the accuracy measure, enabled only during the training phase or periodically.

We tested the proposed methodology with different applications and case studies. The experimental results prove the validity of the approach, provided that the underlying assumption at the base of the employed CDT is satisfied; i.e. the collected data have to be independent of each other and have to originate from the same distribution (i.i.d.). We tested a situation in which this assumption was broken by introducing slight variations in the input data, to which the application was dependent. When the variation is very small, the proposed approach still yields valid results. In another experiment, the data generating application was supposed to provide i.i.d. observations, but the dynamic frequency scaling of the processor of the computer on which the application was being run was introducing huge time dependency, which instead has a dramatic effect, above all on the second level test. We were still able to provide meaningful results by relaxing the statistical robustness of the approach and by taking into account a more empirical effect magnitude of the detected change, by using the Cohen's d test.

Possible future expansions of our work can investigate different improvements:

- Since it is possible for the execution environment to not satisfy the required i.i.d. assumption (sometimes it is enough not having physical access to the server and thus be unable to disable the dynamic CPU frequency scaling technologies in hardware), future works could aim at replacing the Cohen's d test with another approach, able to maintain the statistical guarantees. Otherwise, a very nice and needed research could go towards finding techniques to further relax the above-mentioned assumption; this would make the tool more versatile and easily portable across architectures, which ideally could be treated as black boxes. For instance, High Performance Computing workloads on Amazon Web Services (AWS) are run on virtual servers, known as instances, enabled by Amazon Elastic Compute Cloud (Amazon EC2) [3]. As stated in [15], not every kind of instance has the ability to control such low-level features from the operating system (software) level.

- Considering the proposed tool, we saw that the many exposed parameter controlling the performance of the ICI-based CDT are deeply application dependent. These are the number of subsequences used for the test training phase, the size of the subsequences themselves, and Γ, the amplifying factor of the confidence intervals. It would be a significant contribution to implement a way to carry out a space exploration of these options, tailored for the specific application currently used. Possibly starting from aggressive parameters, which could yield false positives, hopefully rejected by the second level test. If too many FP are observed, automatically the parameters would be gradually relaxed, until a convergence to a good compromise of FP and detection delay is obtained.

- Currently we compute the residuals at the base of the two levels of the hierarchical CDT as the absolute difference between the expected value and the observed one. However, different policies could be enforced according to the actual direction of the change. For instance, if the variance of a process decreases, we could be interested in not raising a detection.

- Metrics like the error can be automatically controlled and enabled either periodically or just during the training phase, or again with a custom command currently never used. The user may be interested, in case of an anomaly detection on an always-enabled metric, in a confirmation of the change also on an usually-disabled metric like the error, before actually triggering a change detection. Thus more advanced policies requiring a more confident detection could be employed. This would require an automated temporary re-enabling of the error monitor, and would introduce a delay in the detection, if confirmed.

Bibliography

[1] America's data centers consuming and wasting growing amounts of energy. https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy.

[2] Apache cassandra homepage. http://cassandra.apache.org/.

[3] Benefits of running hpc on aws. https://aws.amazon.com/hpc/.

[4] Boost program options. https://www.boost.org/doc/libs/1_69_0/doc/html/program_options.html.

[5] Comparing the means of two samples with the students-t test. https://www.boost.org/doc/libs/1_69_0/libs/math/doc/html/math_toolkit/stat_tut/weg/st_eg/two_sample_students_t.html.

[6] Eclipse mosquitto homepage. https://projects.eclipse.org/projects/technology.mosquitto.

[7] Eclipse paho homepage. https://www.eclipse.org/paho/.

[8] Google data centers - efficiency: How we do it. https://www.google.com/about/datacenters/efficiency/internal/index.html#water-and-cooling.

[9] The green 500. https://www.top500.org/green500/.

[10] High-performance computing (hpc). https://searchdatacenter.techtarget.com/definition/high-performance-computing-HPC.

[11] margot core repository. https://gitlab.com/margot_project/core.

[12] Mqtt 101 - how to get started with the lightweight iot protocol. https://www.hivemq.com/blog/how-to-get-started-with-mqtt/.

[13] Mqtt essentials part 9: Last will and testament. https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament/.

[14] Mqtt homepage. http://mqtt.org/.

[15] Processor state control for your ec2 instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html.

[16] Top 500 - november 2018. https://www.top500.org/lists/2018/11/.

[17] Tsukuba dataset. http://cvlab-home.blogspot.com/2012/05/h2fecha-2581457116665894170-displaynone.html.

[18] What is high performance computing? https://insidehpc.com/hpc-basic-training/what-is-hpc/.

[19] What is the environmental impact of a data center? https://www.colocationamerica.com/blog/data-center-environmental-impacts.

[20] Christoph A. Schaefer, Victor Pankratius, and Walter Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. volume 5704, 08 2009.

[21] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. 09 2009.

[22] C. Alippi, G. Boracchi, and M. Roveri. Just in time classifiers: Managing the slow drift case. In *2009 International Joint Conference on Neural Networks*, pages 114–120, June 2009.

[23] C. Alippi, G. Boracchi, and M. Roveri. Change detection tests using the ici rule. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, July 2010.

[24] C. Alippi, G. Boracchi, and M. Roveri. A hierarchical, nonparametric, sequential change-detection test. In *The 2011 International Joint Conference on Neural Networks*, pages 2889–2896, July 2011.

[25] C. Alippi and M. Roveri. Just-in-time adaptive classifiers-part i: Detecting nonstationary changes. *IEEE Transactions on Neural Networks*, 19(7):1145–1153, July 2008.

[26] C. Alippi and M. Roveri. Just-in-time adaptive classifiers-part ii: Designing the classifier. *IEEE Transactions on Neural Networks*, 19(12):2053–2064, Dec 2008.

[27] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[28] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: Online autotuning through local competitions. pages 91–100, 10 2012.

[29] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 45(6):198–209, June 2010.

[30] Blaise Barney. Introduction to parallel computing. `https://computing.llnl.gov/tutorials/parallel_comp/`.

[31] Michèle Basseville and Igor V. Nikiforov. *Detection of Abrupt Changes - Theory and Application*. Prentice Hall, Inc. - http://people.irisa.fr/Michele.Basseville/kniga/, 1993.

[32] Michèle Basseville and Igor V. Nikiforov. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[33] I.W. Burr. *Statistical Quality Control Methods*. Statistics: A Series of Textbooks and Monographs. Taylor & Francis, 1976.

[34] Matteo Cesana. Un framework teorico e tecnologico per sistemi ciberfisici intelligenti. Master's thesis, Politecnico di Milano, 2015.

[35] L.H. Chiang, E.L. Russell, and R.D. Braatz. *Fault Detection and Diagnosis in Industrial Systems*. Advanced Textbooks in Control and Signal Processing. Springer London, 2012.

[36] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Taylor & Francis, 2013.

[37] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005.

[38] D. Gadioli, G. Palermo, and C. Silvano. Application autotuning to support runtime adaptivity in multicore architectures. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 173–180, July 2015.

[39] Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Cristina Silvano. margot: a dynamic autotuning framework for self-aware approximate computing. *IEEE Transactions on Computers*, 2018.

[40] Y. Gao and P. Zhang. A survey of homogeneous and heterogeneous system architectures in high performance computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 170–175, Nov 2016.

[41] Giacomo Boracchi. Change detection tests using the ici rule. `http://home.deib.polimi.it/boracchi/docs/2010_08_31_ICI_for_Change_Detection.pdf`.

[42] A. Goldenshluger and A. Nemirovski. On spatial adaptive estimation of nonparametric regression. *Math. Meth. Statistics*, 6:135–170, 1997.

[43] R J Hernstein, D H Loveland, and C Cable. Natural concepts in pigeons. *Journal of experimental psychology. Animal behavior processes*, 2 4:285–302, 1976.

[44] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. *SIGPLAN Not.*, 45(5):347–348, January 2010.

[45] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. *SIGPLAN Not.*, 46(3):199–212, March 2011.

[46] V. Katkovnik, K. Egiazarian, and J. Astola. A spatially adaptive nonparametric regression image deblurring. *IEEE Transactions on Image Processing*, 14(10):1469–1478, Oct 2005.

[47] Ken Kelley and Kristopher J. Preacher. On effect size. *Psychological Methods*, 17(2):137–152, 2012.

[48] M.G. Kendall and J.D. Gibbons. *Rank Correlation Methods*. Charles Griffin Book. E. Arnold, 1990.

[49] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[50] D. M. Kunzman and L. V. Kale. Programming heterogeneous systems. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 2061–2064, May 2011.

[51] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: Using canary inputs to dynamically steer approximation. *SIGPLAN Not.*, 51(6):161–176, June 2016.

[52] J. Lerga, M. Vrankic, and V. Sucic. A signal denoising method based on the improved ici rule. *IEEE Signal Processing Letters*, 15:601–604, 2008.

[53] Bryan F. J. Manly and Darryl Mackenzie. A cumulative sum type of method for environmental monitoring. *Environmetrics*, 11(2):151–166, 2000.

[54] Cristiano Di Marco. A distributed framework supporting runtime autotuning for hpc applications. Master's thesis, Politecnico di Milano, 2017.

[55] Joshua San Miguel and Natalie Enright Jerger. The anytime automaton. *SIGARCH Comput. Archit. News*, 44(3):545–557, June 2016.

[56] R.G.J. Miller. *Simultaneous Statistical Inference*. Springer Series in Statistics. Springer New York, 2012.

[57] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 25–34, May 2010.

[58] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[59] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, March 2016.

[60] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[61] Seyed Mostafa Mousavi Kahaki and Md Jan Nordin. Highway traffic incident detection using high-resolution aerial remote sensing imagery. *Journal of Computer Science*, 7:949–953, 05 2011.

[62] Govind S. Mudholkar and Madhusudan C. Trivedi. A gaussian approximation to the distribution of the sample variance for nonnormal populations. *Journal of the American Statistical Association*, 76(374):479–485, 1981.

[63] E. Paone, D. Gadioli, G. Palermo, V. Zaccaria, and C. Silvano. Evaluating orthogonality between application auto-tuning and run-time resource management for adaptive opencl applications. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 161–168, June 2014.

[64] Edoardo Paone, Gianluca Palermo, Vittorio Zaccaria, Cristina Silvano, Diego Melpignano, Germain Haugou, and Thierry Lepley. An exploration methodology for a customizable opencl stereo-matching application targeted to an industrial multi-cluster architecture. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 503–512, New York, NY, USA, 2012. ACM.

[65] Ian Ritchey. Automating change detection with exogenesis. `https://www.planet.com/pulse/automating-change-detection-with-exogenesis/`.

[66] Graeme D. Ruxton. The unequal variance t-test is an underused alternative to Student's t-test and the Mann-Whitney U test. *Behavioral Ecology*, 17(4):688–690, 05 2006.

[67] Rowayda Sadek, Mohamed Soliman, and Hagar S. Elsayed. Effective anomaly intrusion detection system based on neural network with indicator variable and rough set reduction. *International Journal of Computer Science Issues*, 10:227–233, 11 2013.

[68] Shlomo S. Sawilowsky. New effect size rules of thumb. *Journal of Modern Applied Statistical Methods*, 8(2):597–599, nov 2009.

[69] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. Proactive control of approximate programs:. *ACM SIGARCH Computer Architecture News*, 44:607–621, 03 2016.

[70] R. K. Tripathi, A. S. Jalal, and C. Bhatnagar. A framework for abandoned object detection from video surveillance. In *2013 Fourth National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG)*, pages 1–4, Dec 2013.

[71] Scott E. Umbaugh. *Digital Image Processing and Analysis: Human and Computer Vision Applications with CVIPtools, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2010.

[72] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[73] K. Zhang, J. Lu, and G. Lafruit. Cross-based local stereo matching using orthogonal integral images. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(7):1073–1079, July 2009.

[74] Donald W. Zimmerman. A note on preliminary tests of equality of variances. *British Journal of Mathematical and Statistical Psychology*, 57(1):173–181, 2004.