On Estimating Actuation Delays in Elastic Computing Systems

Alessio Gambi*†, Daniel Moldovan*, Georgiana Copil*, Hong-Linh Truong*, and Schahram Dustdar*

*Distributed Systems Group, Vienna University of Technology, Austria, {lastname}@dsg.tuwien.ac.at
† University of Lugano, Switzerland, {name.lastname}@usi.ch

***Abstract*—Elastic controllers autonomically adjust the allocation of resources in cloud computing systems. Usually such controllers assume that control actions will take immediate effect. In clouds, however, actuation times may be long, and the controllers can hardly guarantee acceptable levels of service if they neglect these actuation delays. Therefore, the ability to correctly estimate the time that control actions take effect on the systems is crucial. However, detecting actuation delays in elastic computing systems is challenging because cloud systems provide only inaccurate and incomplete data about reconfigurations timing.**

In this paper, we tackle the problem of estimating the delay of control actions in elastic systems. We identify recurring types of changes in the monitored metrics and requirements to properly carry out the estimation. Based on that, we develop a novel framework for the actuation delays estimation that utilizes change point detection techniques. We conduct several experiments with real-world systems to illustrate the feasibility and applicability of our framework.

***Index Terms*—Elasticity, cloud, system identification, change point detection**

I. Introduction

In order to be aligned with the fluctuations of the incoming workload elastic cloud systems acquire fresh computational resources when the load increases and release them when the load decreases. By dynamically optimizing the total amount of acquired resources, elastic cloud systems can provide consistent quality of service while minimizing costs.

This paper targets elastic systems that are implemented by coupling a dynamically adaptive system (DAS) with a controller in a closed-loop style to provide self-* properties, like self-optimization of resources [4]. The controller implements the so called MAPE-K loop [14] by repeatedly performing the following activities: monitor the main variables of the DAS, like the incoming load, the average usage of resources, and the system performance; analyze the monitoring data to identify trends or risky situations that may lead to unacceptable system behaviors, for example, degraded end-to-end performance; plan a control strategy to provide the required level of service; and eventually, implement the control actions by means of the actuators.

In particular, we examine the case where the DAS is a virtualized system that runs in a cloud and contains all the required logic to manage resource acquisition and release. The controller uses the available cloud APIs for monitoring the virtualized system and implementing the control actions, i.e., adding or removing virtual machines (VMs) at runtime. Examples of such systems can be found in [19], [22], [28].

A. Motivation

Currently, the above-mentioned elastic systems are designed under the following assumptions: (i) they can compute a control strategy within their control loop; (ii) the implementation of the control actions is immediate; and (iii) the effects of the control actions on the system behavior appear with no delay. Ideally, the controller analyzes the monitoring data, determines a suitable control strategy, and triggers the proper actions in every and all control cycles. In turn, control actions complete and take effect before the next cycle begins.

To date, existing research has focused on optimizing controllers to make the first assumption hold, i.e., find an optimal solution fast enough, while the other two assumptions are satisfied "by design": Either controllers are slowed down such that control actions appear to be immediate, or they are inhibited between consecutive control actions by means of hystereses [3]. This approach becomes critical in elastic systems where actuation times may be longer than expected. In fact, cloud platforms may take a relatively long time to deploy virtual machines [18]. And after that, virtual machines may need additional time to boot, configure their application components, and eventually, join the elastic systems [23].

In this situation it is very difficult, or even impossible, for traditional elastic controllers to maintain the expected quality of service at the expected costs. In fact, when the input workload changes at a faster pace than the system, control actions will be late and this may lead to unstable behaviors and resource *thrashing* [8]. For example, consider the case that the incoming load increases and the elastic system asks for additional resources. While resources are provisioned the incoming load may vary: If it increases further, in the next control cycle the system will be under-provisioned, and the controller will add even more resources. Instead if the incoming load decreases, in the next control cycle the system will be over-provisioned, and the resources just added will be soon removed.

One can improve controllers by letting them work at a faster pace and by explicitly taking into account the time each control action takes to complete, i.e., its actuation delay. Two questions arise in this case: i) How can controllers and their designers exploit this knowledge to improve the quality

978-1-4673-4401-2/13/$31.00 © 2013 IEEE

SEAMS 2013, San Francisco, CA, USA

of control?, and, ii) How can actuation delays be properly measured, estimated and modeled?

Several researchers investigated the first question. For example, Ramirez et al. [24] propose the use of Genetic Algorithm to generate control strategies that account for actuation delay of the different control actions; Fritsch et al. [10], [11] propose a time-bounded scheduling algorithm for control actions in the automotive domain; and Gambi et al. [12] propose a novel technique based on model predictive control that accounts for delayed effects of scaling actions in the cloud. Consistently, all of these researchers acknowledge the importance of actuation delays, but assume that their values are known and constant. To our best knowledge however there is a lack of solutions for building accurate models to predict actuation delays that controllers, especially in elastic systems, can seamlessly integrate in their control loops.

B. Research Problems and Contributions

In this paper, we focus primarily on the problem of estimating the actuation delays for elastic systems, as accurate estimations are the foundation for building the reliable models that controllers need to improve their effectiveness. The estimation of actuation delays in elastic systems is challenging for several reasons. First, there is no single, reliable and easily measurable metric that can be used to estimate the time that changes in elastic systems take to complete. Cloud systems provide users with information about the current system configuration in terms of the number of virtual machine deployed, their status, and their type. But such data may be incomplete and inaccurate: They are inaccurate because cloud systems notify changes in the system configuration when physical resources are allocated to the VM, even before the OS starts. They are incomplete because cloud systems are not aware of the application components inside the VMs, therefore they cannot tell when and if those components are ready to process any request.

Second, the effects of adapting an elastic system may appear with a variable delay on its behavior depending on the type of adaptation and the current state of the system. For example, when an elastic database system scales out its persistence tier, the delay may depend on the actual distribution of the data among the databases and the number of databases running.

Third, the same adaptation may result in different effects across system properties and each of them may be observed with a different delay. For example, when a new server is added to the Web tier the effects on the system performance can be quickly observed, while the effect on system availability may be visible only later. As a result, it is difficult to select a proper metric (or a limited set of metrics) that must be monitored to understand when the action starts and when it is over.

Finally, clouds are shared between several users, and this may result in subtle interferences that create transitory, yet abnormal, deviations from the normal behavior. These can be misinterpreted as effects of system changes, and make the estimation difficult. They require a proper treatment, otherwise the overall accuracy of the estimation may degrade.

In our approach, we observe the behavior of the controlled systems through the available monitoring infrastructure, as designers would do during system identification or controllers do at runtime. We estimate the actuation delays by analyzing how system reconfigurations affect the evolution of the system behavior, and we focus on finding the precise moment when reconfigurations are over, i.e., when all their effects appear on the observed metrics.

This is in fact the information that a controller needs in order to decide when it is safe to schedule the next control action such that it will not interfere with the ones already submitted, and vice versa. To this end, this paper makes the following contributions:

- We provide a detailed analysis of the problem of estimating the delay of control actions in elastic systems, covering recurring types of changes in the monitored metrics and requirements to properly carry out the delay estimation.
- We develop a novel framework for the estimation of actuation delays and a prototype implementation.
- We conduct an empirical evaluation about the feasibility and applicability of the proposed framework with two real-world case studies.

The remainder of the paper is organized as follows: Section II presents a detailed analysis on the problem of delay estimation in elastic cloud systems. Section III describes the design and implementation of our delay detection framework. Section IV evaluates our techniques with real case studies. Section V discusses the relate work. Finally, Section VI concludes the paper and presents the future work.

II. Analysis of Actuation Delays in Elastic Systems

A. Reference Model for Delay Estimation

To ease the analysis of problems and requirements for the estimation of actuation delays, we introduce the relevant concepts related to actuation delays:

- *Action*. In the context of the considered elastic systems the concept of control action, or reconfiguration action, identifies the controller's ability to change the system resources allocation by starting or terminating instances of virtual machines.
- *Actuation time*. Actuation time measures the time that actuators need to implement a reconfiguration action, that is, the time that the cloud needs to deploy or terminate VMs. Actuation time accounts only for the time to reconfigure the system at the infrastructure layer, hence it does not account for the time to boot or shutdown virtual machines and to configure the system at the application layer.
- *Action effect*. Action effects are modifications in the operation parameters of the system that derive from the application of a control action.

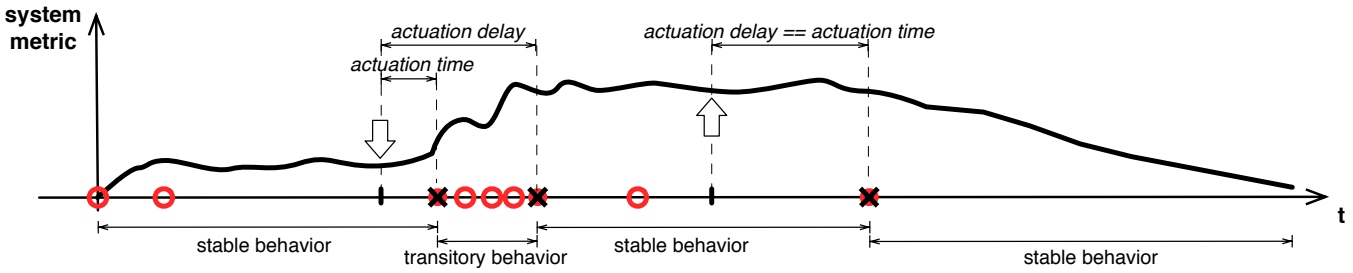

Fig. 1. Analysis of changes in system metric for delay estimation

- *Metric*. Metrics are measurable properties of a system. Action effects can be observed through metrics, therefore we study these changes to characterize them.
- *Actuation delay*. The focus of this work is to analyze the actuation delays, i.e., the amounts of time needed for control actions to show their effects on the system behavior. Actuation delays account for the time to implement changes at the infrastructure level, i.e., the actuation time, for the additional time to reconfigure the elastic system at the application level, and the time to observe the changes of the monitored metrics. Changes at the infrastructure level depend on the control action and on the cloud platform, observation delays depend on the monitoring framework and observed metric, while application-level reconfigurations may depend on the actual operational point of the application itself, especially if they entail data replication and distribution. In this work, we assume that reconfigurations introduce a disturbance in the system metrics that has limited time-span; in other words, we assume that the system eventually reaches a stable state.

For the sake of explanation, we illustrate the above-mentioned concepts over a realistic example. Figure 1 shows the evolution of a system metric (solid line) in time, while the system is subjected to input disturbance (not showed in the figure) and changes in the system configuration (white arrows). We highlight important changes of the output metric by means of (red) circles. They reflect on the metric changes in the system behavior that may be caused by background operations, interferences, environmental variability, changes in the inputs, and the application of reconfiguration actions. To estimate the delay of reconfiguration actions, we are interested in finding only those changes that are caused by the actions and that we highlight with (black) crosses in Figure 1.

B. Classifying Action-Effect Types

Depending on the system under investigation, the observed metric and the applied action, we can see different effects, i.e., the metric shows a different form after the application of the action. During our analysis of elastic systems, we noticed that when similar actions are applied to the system under a constant workload the output metrics change in similar ways, even if the system had configurations before the control action changed it. We use these recurring patterns to compile an initial classification of changes that is exemplified in Figure 2, and that we briefly discuss in this section[1].

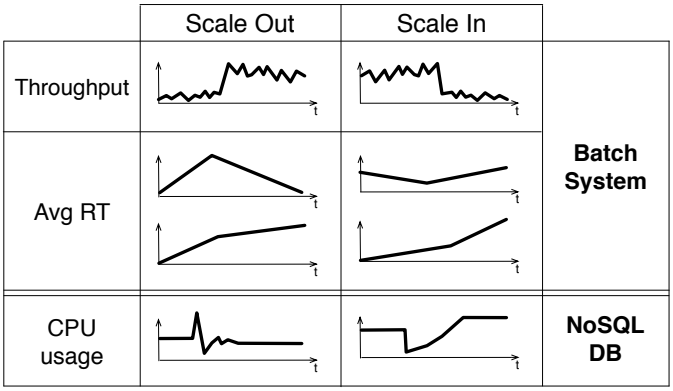

Fig. 2. Examples of changes in system metrics after reconfigurations.

In Figure 2, we group reconfiguration actions into `Scale Out` and `Scale In` actions where `Scale Out` identifies the addition of new resources, and `Scale In` indicates the removal of resources. We then group patterns by metrics and types of elastic systems. At the time of writing, we considered two types of elastic systems widely used in clouds, Batch Systems and NoSQL Databases; and three metrics that are commonly used to characterize their performance: Throughput, average response time (Avg RT), and CPU usage. For batch job processing systems, throughput and average RT define the end-to-end qualities that elastic systems need to provide by acting upon their configuration [12]. For NoSQL databases instead, CPU usage is the reference metric considered by elastic controllers it being the one that impacts the most the system behavior [17].

Figure 2 shows how these metrics evolve after the application of scaling actions:

- *Batch system throughput*: When more resources are added to the system (`Scale Out`) the throughput increases and stabilizes, while when resources are removed it decreases and stabilizes.
- *Batch system average response time*: For the `Scale Out` action, we see that when the system

[1]We provide supplement materials at http://www.infosys.tuwien.ac.at/prototypes/elasticTesting/index.html for additional details about the classification and experiments.

is under-provisioned, i.e., the average response time is constantly increasing for a constant load, adding more resources may end up in two cases: The response time starts decreasing constantly; this happens if enough resources are added to the system. Otherwise, the response time keeps increasing with a slower pace. For the `Scale In` action we have the opposite situations.
- *NoSQL DB CPU Usage*: After a `Scale Out` we see that the CPU usage has a spike, and after a transitory period, it stabilizes. While after a `Scale In` instead it initially drops almost to zero, and then slowly recovers to its new value.

This classification captures the expected system response to different stimuli and is important because it reduces the complexity of identifying what changes can be observed in the output metrics by clustering the effect of different actions in predefined classes. Currently, our classification contains a total of fifteen patterns. We use this knowledge to improve the estimation process by parameterizing it with respect to the expected change class: By knowing what is the expected type of change we may select a particular algorithm that is proven to be more accurate for detecting that particular type of change. For example the throughput shows strong oscillations around a constant mean, hence we may favor an algorithm that is capable of detecting changes by checking variations of the mean.

C. Requirements for Accurate Delay Estimation

To perform an accurate estimation of the actuation delays we need to capture the important changes in the system metrics. This requires having a way to analyze every single metric to detect when and if a change happens. This also requires that we apply the analysis to the right set of metrics because different metrics may be impacted differently, and some of them may show no impact at all. Therefore it is critical to discriminate when a metric is impacted in a important way by a reconfiguration action. Overall, we have identified the following important requirements:

Ability to deal with noise. Metrics can be corrupted with noise and the analysis of these metrics may identify several changes, many of which are just an effect of the noise. To improve the quality of the analysis, on the one hand, we need to understand when we should reject the analysis due to unfavorable ratio of system metrics to noise, and when we should pre-process metrics by filtering or smoothing them.

In case the system runs in a dynamic environment, this analysis also needs to adapt to the current working conditions and evolving context.

Improvement of estimation confidence by combining different data. It may happen that too many changes are still present in the metric. In this case, we need a way to combine the data about the delays among several metrics to gain confidence on the time of beginning and ending of the actions. This involves the definition of correlation analysis to capture the delays across the metrics as well as the definition of aggregation functions to combine the observations across several measurements of the same metric, system and action.

Use of contextual information. Contextual information about the environment and the disturbance on the system, i.e., its input workload, should be considered when analyzing the identified changes. This may highlight correlations among changes in the input metrics (or environment) and changes in the output metrics of the system that enable the identification of the changes caused by the reconfigurations and the ones caused by the input variability.

III. Delay Estimation Framework

A. Using Change Point Detection for Delay Estimation

Broadly speaking, change point detection (CPD) can be considered as the process of identifying points within a data set where the statistical properties change [25]. It arose as a mechanism for addressing shortcoming in time series analysis techniques that assume stationarity, i.e., parameters that describe the properties of the dataset do not depend on time [2]. CPD is proven to be successful in many practical problems such as anomaly and fault detection [27], black-box Web service reliability monitoring [9], and detection of security attacks [30].

In this work, we argue that CPD can be also used to estimate actuation delays in the context of elastic systems. By analyzing the system metrics after a reconfiguration, CPD can produce a list of potential changes that are related to it. In other words, under the assumption that the available output metrics define the system behavior, we argue that a reconfiguration changes the behavior of the system and that we can identify the precise moment when this happens by capturing the changes in such output metrics. Consequently we can estimate the time it took to complete the reconfiguration and to reach the next stable state.

By adopting CPD we can fulfill the requirements on detecting changes in the system metrics and discriminate which metrics are impacted by any given control action.

B. Estimation Process

The estimation of the actuation delays can be seen as an activity belonging to system identification [21]. Therefore we instantiate the standard system identification process for our specific case. We design a gray-box system identification process where we can observe both inputs and outputs of the system, as well as its current configuration. Furthermore, we can control the inputs of the system, including the ones that generate the disturbance by injecting loads.

We depict the main activities of the estimation process for actuation delays and their organization in Figure 3. The process is articulated in the following points: 1) Setup the environment and prepare the experiment; 2) Deploy the system in the cloud; 3) Put the system under stress and wait until it enters a stable state; 4) Actuate the system reconfiguration and wait; 5) Tear down the environment; 6) Store the monitoring data; and, 7) Analyze the monitored data to estimate the actuation delay.

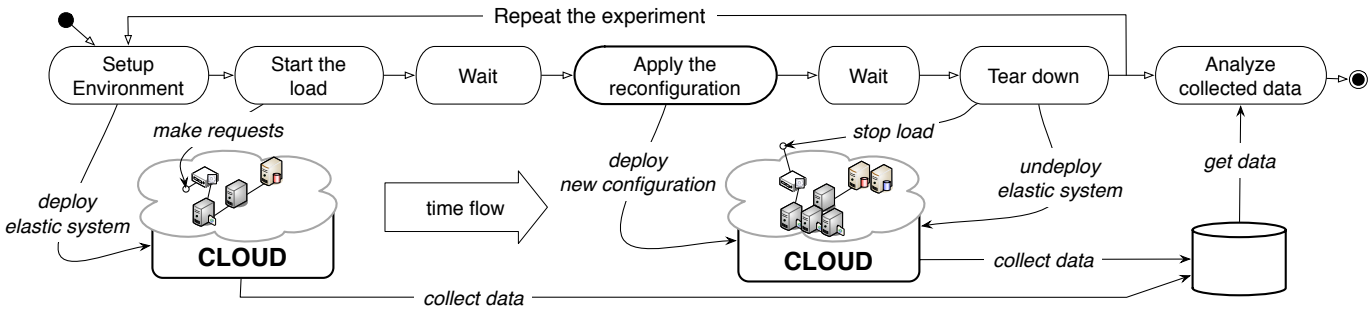

Fig. 3. Main activities of the delay estimation process

By iterating this process several times, we can collect additional data and gain confidence about the obtained results, thus addressing the relative requirement. For the analysis we input into the estimation framework the collected metrics about the system within the data about its initial configuration, the acted reconfiguration, and the expected outcome in terms of the number of change points. The framework then runs the CPD algorithms on the main metrics and may use additional information to filter out the spurious change points. When multiple runs are available, we can combine their results to provide the final actuation delay estimation.

C. Estimation Framework

We design a framework that leverages experiment automation to reduce the burden of conducting the estimation process and to speed up its execution. We depict the conceptual architecture of the framework in the left-side of Figure 4. In the figure blocks identify the main components and arrows model the flow of data. The *system under investigation* exposes a *control interface* to change its configuration and (if available) a *user interface*. The control interface is where a controller would bind at runtime, while the user interface is where final users of the elastic system place the workload. The framework provides two components that use these interfaces to stimulate the system under investigation: The *action enforcer* is the driver for the estimation process and is in charge of triggering system reconfigurations by invoking the control interface. The *stress generator* represents the environment and controls the disturbance on the system by interacting with the final user interface. While performing an experiment the framework records the triggered reconfigurations and the disturbance, as well as the system metrics. It reports everything to the *actuation delay estimator* that carries out the analysis to correlate monitoring data and estimate the actuation delay.

The right side of Figure 4 details the actuation delay estimator and shows the flow of activities and data inside it. The evolution of the system metrics over the experiment is analyzed by the change point detection block, and a list of potential change points for each of them is produced. We represent change points in the plots as (red) circles. The CPD block may run different change point detection algorithms or the same algorithm with different parametrization to produce the list of change points. The choice of the algorithm depends on the type of actions under study and the effects that are expected to be captured by the action-effect classification. We discuss issues about choosing the proper algorithms and their configurations in the next section.

The list of change points and the data about the reconfiguration action are then passed to the *filtering and correlation* block that removes spurious change points and estimates the actuation delay. This block, for example, removes all the change points that are identified before the application of the reconfiguration, and can filter the change points that appear while the system is reconfiguring. Furthermore, it may correlate change points detected across different metrics to understand when the system reaches a stable state.

D. Estimation Algorithms

As mentioned, we apply change point detection algorithms for detecting actuation delays. CPD algorithms can be classified in offline and online algorithms: Offline algorithms consider whole time series at once to find the possible change points. Online algorithms consider data incrementally and discover change points as they appear.

We have evaluated the use of CPD algorithms in a process where data analysis is performed in batch and offline. Based on the capability of evaluated algorithms, we select two state-of-the-art algorithms, one for each family, for our framework:

- **Pruned Exact Linear Time (PELT)** [16]: PELT belongs to the family of offline change point detection algorithms and works by splitting the data in homogeneous partitions. Among all the possible splits, the algorithm chooses the one that minimizes a given cost metric by exploiting dynamic programming to carry out the optimization. PELT can be applied to identify variations in the mean of a metric, in its variance, or in both.
- **Bayesian Online Change Point Detection (BOCPD)** [29]: BOCPD can be used for online, adaptive change point detection. The algorithm exploits the Bayesian theory to estimate the probability of having a change in the next sample, given the sequence of observations collected so far. In particular, the algorithm uses a prediction model based on Gaussian processes that is combined with an hazard function to compute the

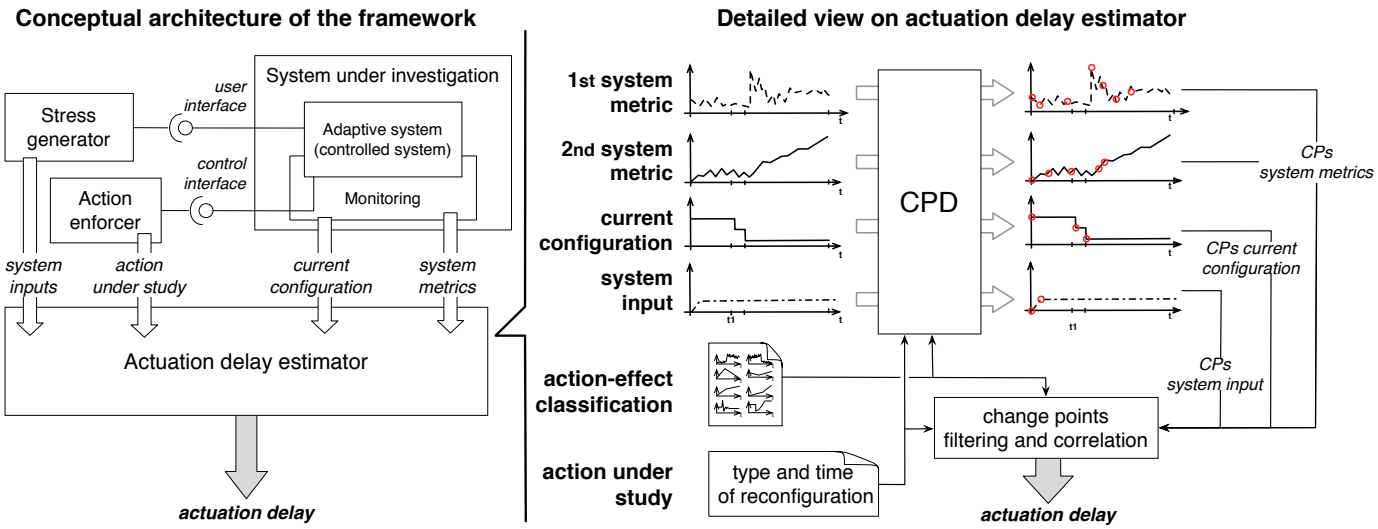

Fig. 4. Conceptual architecture of the delay estimation framework

probability of the change. The model is retrained after the identification of a change point to adapt to the new conditions.

We use PELT to find changes when the system switches between steady states, that is, where the mean of the metric is constant before and after the reconfiguration. Otherwise, if the system reaches stable states where the mean value of the metrics is not steady, then the algorithm will identify many more change points. Instead, BOCPD is not tied to any particular statistical moment of the analyzed metric and is more generally applicable. For example, it can be suitably used when the system moves between stable states, either that they show constant mean or not. However, BOCPD is sensitive to frequent oscillations and noise in the metric, therefore it is difficult to use under these conditions as the algorithm may detect too many changes.

IV. Evaluation

We have implemented Java components for controlling the experiments and we have integrated existing libraries for generating the load and monitoring the system metrics. We experimented our prototype with two different cloud platforms, namely OpenStack[2] and Reservoir[3]. We analyzed monitoring data using the available implementations of PELT, written in R[4], and BOCPD, written in Matlab[5].

We conducted several experiments to answer the following questions: Can CPD algorithms identify the changes that are caused by reconfiguration actions? What types of changes each of the selected CPD algorithms can identify? Is the proposed approach able to estimate actuation delays?

[2]http://www.openstack.org/
[3]http://www.reservoir-fp7.eu/
[4]http://www.r-project.org/
[5]http://www.mathworks.ch/products/matlab/

A. Case Studies

To evaluate our framework we ran several experiments against two real elastic systems that are commonly run in clouds: Cassandra[6], a NoSQL distributed database, and Sun Grid Engine (SGE)[7], a grid middleware. We used SYBL [7] for controlling elasticity (i.e. scaling in/out considering time and loads).

1) The Cassandra NoSQL Database: Cassandra is an adaptive distributed data store that can accommodate the addition and the removal of processing nodes at runtime. In our setup, we deploy a Cassandra cluster with a single Cassandra seed node that acts both as system access point and cluster controller. The Cassandra seed distributes the incoming workload across the processing nodes. When a processing node joins or leaves the cluster, the Cassandra seed updates the load distribution and if necessary it copies or moves data across the remaining processing nodes. In this evaluation, we use the default logic of Cassandra for the load distribution and data management

We installed the controller and the processing nodes in separate virtual machines that we deployed in our private OpenStack cloud. All the virtual machines contain the logic to automatically join and leave the system. Hence we act only at the infrastructure level to reconfigure the elastic system: We start new instances of the processing node to scale out the system, and we terminate them to scale the system in. New instances of the processing node are placed by the cloud middleware on physical servers, and, after the VMs boot, processing nodes register to the controller that, in turn, reconfigures the load distribution and the data placement. Only after all these operations complete, the new processing nodes can start to serve requests. When we terminate instances of the

[6]http://cassandra.apache.org/
[7]Now called Oracle Grid Engine and available at http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html

processing node, the cloud middleware stops abruptly their virtual machines without notifying the Cassandra controller. The controller notices that processing nodes are down because they do not respond anymore, and only after that, it adapts the load distribution and the placement of data. While doing this operation Cassandra will stop servicing new requests until it determines where to forward the requests that address the just removed processing nodes.

To monitor the virtual infrastructure we use Ganglia[8] that gathers several low level metrics at fine grained intervals. We use our prototype to aggregate data across the virtual machines. We stress the elastic system with the Yahoo! Cloud Serving Benchmark (YCSB) [6].

2) The Sun Grid Engine: SGE is a grid middleware for the distributed processing of jobs submitted by final users. It follows a standard architecture with a singleton master node and a set of executor nodes: The master receives jobs from clients and dispatches them to the executor nodes to be run. We use the implementation of SGE that was delivered in the Reservoir EU project that implements elasticity at the level of executor nodes.

We installed the master and executor nodes into two types of virtual machines that we deployed in the Reservoir cloud. As before, we control the system at the infrastructure level by acting on the instances of virtual machines. New instances of the executor register with the master after their virtual machines boot, and deregister from it when capture the shutdown OS-level signals that the hypervisor triggers to terminate them.

To monitor the virtual infrastructure we use the Lattice framework [5]. We stress the application using the Apache JMeter [9] load generator.

B. Experiment Design

We ran several experiments according to the estimation process presented in the previous section, and we analyzed system metrics with CPD algorithms to produce the list of change points. We then elaborated this list to estimate the actuation delays.

Experiments that investigated Cassandra started with an initial interval of 30 minutes to let the system stabilize against the input workload. Next, either a single `Scale Out` or `Scale In` operation was triggered. After this, we kept the system running for another thirty-minute period to stabilize before ending the experiment run. For the change point detection, we selected the CPU usage metric. We made this choice because Cassandra's operations are in general CPU bound and this metric provides a good indicator about the system workload. Before running the CPD analyses we aggregated the collected data over 30 second-long non-overlapping time-frames by computing the average value of the metric over each time-frame.

Experiments that investigated SGE started with an initial interval of 10 minutes where the system was subjected to a stable input workload. Then we applied the reconfiguration action and kept the system running for another ten-minute period before ending the experiment run. For the analysis, we selected the throughput and response time metrics as provided by the monitoring sub-system, that is, without performing any pre-processing on them.

C. Experiment Results

Here we provide a detailed description only of a fraction of the results that we obtained in both the case studies. The entire dataset is available in our companion Web site.[10]

We start by analyzing a `Scale Out` operation that deployed a new processing node to Cassandra. Figure 5 shows the evolution of the average CPU usage and CPU usage while performing disk I/O. The vertical line identifies the instant of action triggering, the circles identify the change points detected by PELT and the crosses the ones identified by BOCPD. PELT detected two change points: The first corresponded to a steep increase in the CPU usage and the second marked when the system reached the steady state. We associated the spike in the CPU to the start of `Scale Out` operation while the steady state with its end. As expected, the average CPU usage after the operation decreased as the load is distributed across an increased number of processing nodes.

We cross-compared the CPU usage and CPU usage while performing disk I/O operations collected during the experiment to verify this: We can see that both metrics increased and decreased consistently, an effect that we explain by knowing that Cassandra is copying data to the new node. We computed the time difference between these two instants to obtain an estimation of the actuation delay of circa 180 seconds. For the same dataset BOCPD detected more change points. This happened because we ran BOCPD with no training points to leverage its ability to adapt to the current system behavior. Some spurious change points were detected while the algorithm had limited knowledge of the dataset. Once the algorithm learned the model, no more change points were identified until the reconfiguration occurred. We discuss this point in the next section. After the reconfiguration, BOCPD was able to detect the two critical changes. To identify the critical change points, we filtered out the spurious ones because they appeared before the reconfiguration was triggered. The estimation of the actuation delay was consistent with the one provided by PELT.

Figure 6 shows the evolution of Cassandra CPU usage after the application of a `Scale In`. The operation removed a processing node from the initial cluster. In this case, PELT identified five change points while BOCPD identified four change points. We associated the drop of the CPU with the start of the reconfiguration, therefore we considered the first change point for the estimation. We can explain this behavior by recalling that Cassandra stops servicing requests when nodes are removed until it reorganizes the data. We associated the steady state after the drop to the end of the reconfiguration,

[8]http://ganglia.sourceforge.net/

[9]http://jmeter.apache.org/

[10]http://www.infosys.tuwien.ac.at/prototypes/elasticTesting/index.html

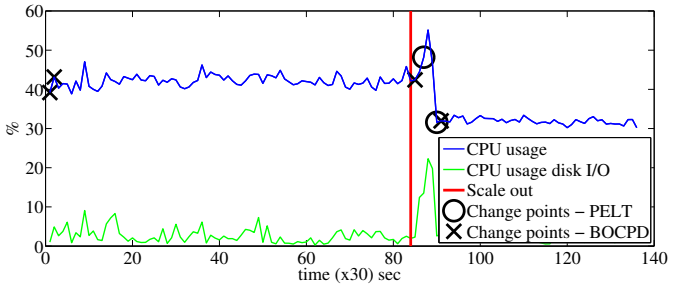

Fig. 5. Change points of Cassandra average CPU usage after a scale out

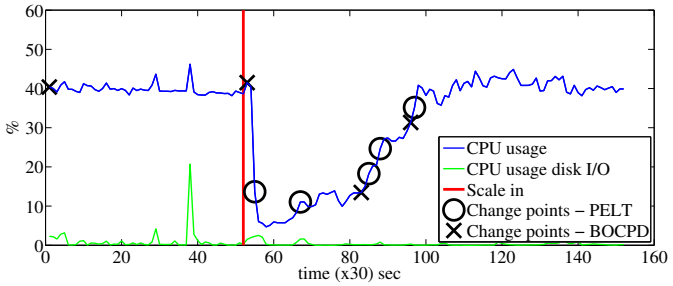

Fig. 6. Change points of Cassandra average CPU usage after a scale in

therefore we considered the last change point that was detected as end of the reconfiguration. The average CPU usage had only a little increase after the reconfiguration because in the default settings Cassandra randomly distributed the load across processing nodes, and in this experiment the processing node that was removed was lightly loaded. Nevertheless, we verify that the system throughput decreased after the `Scale In` by inspecting the number of packets sent out by Cassandra. Both of the algorithms identified the drop in the CPU usage and detected several changes before the system entered in a steady state. We then computed the delay that amounted to circa 420 seconds. Considering only the first and last change point after the reconfiguration both the CPD algorithms performed similarly.

Results for the SGE case study in terms of average response time (top plot) and throughput (bottom plot) for both operations are reported in Figures 7 and 8.

Figure 7 shows the `Scale Out` that took the system from two to four nodes. In this case, both PELT and BOCPD found two change points in the throughput: PELT found a change point right after the reconfiguration and one at the end of the run, while BOCPD found one at the first sample and one after the reconfiguration. We removed the very first change point and the very last because they are a side-effect of the algorithms. SGE does not require any reconfiguration but a file update to accommodate the new nodes, therefore the effects of the scale out were immediate, i.e., the reconfiguration took place with no evident transitory period between the steady state of the system. We estimated the actuation delay by considering the only change point detected and we obtained a value of circa 120 seconds for PELT and circa 145 seconds

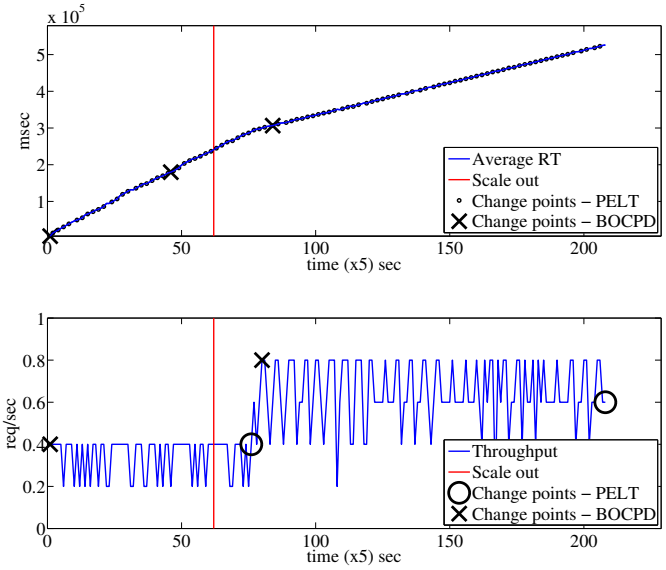

Fig. 7. Change points of SGE throughput and response time after a scale out

for BOCPD. We applied the CPD algorithms also the average system response time. In this case PELT identified too many change points, meaning that it was not able to correctly capture the changes in the response time metrics. At the contrary, BOCPD identified only three change points. For BOCPD, we filtered out the change points that were captured before the reconfiguration took place. We used the remaining one to compute the actuation delay, and verify that this second evaluation was consistent with the previous one. By combining the evidences we were able to confirm the estimated value of the delay of circa 145 seconds.

Figure 8 shows a similar situation after a `Scale In` that took the system from two executor nodes to one. Both PELT and BOCPD found two change points in the throughput, half of which were spurious, and therefore, removed. According to the remaining change points we estimated the actuation delay of about 110 seconds using PELT and about 150 seconds using the BOCPD. We ran PELT and BOCPD on the average response time and we found again that PELT was not able to detect the right change points. BOCPD instead found the change point after the reconfiguration. Differently than before, this second estimation was more consistent with the one by PELT for the throughput, and we estimated the actuation delay for the `Scale In` operation of about 115 seconds (the mean value between the observations).

D. Discussion of the Results

In the investigated cases we observed that change point detection can effectively detect changes that were introduced by the reconfiguration actions. Hence, CPD can be used as an effective means to estimate actuation delays in elastic systems. We saw also that not all the considered CPD algorithms can detect all the changes that we generated on the various system metrics. Furthermore, the two algorithms can provide consistent but slightly different results when they are applied

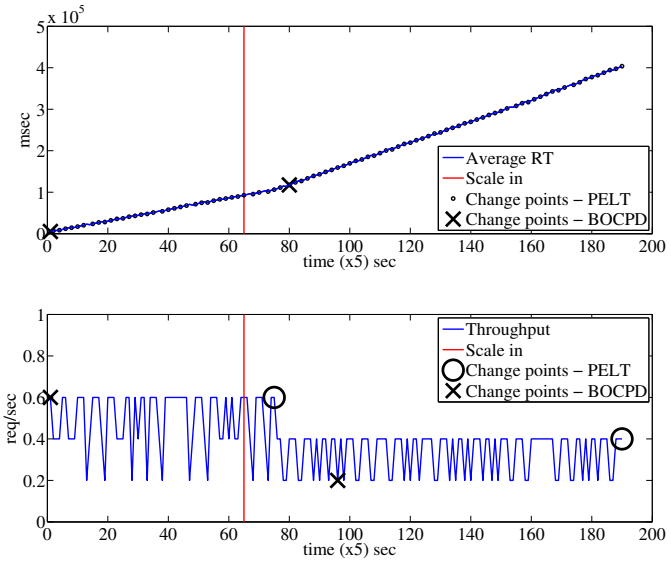

Fig. 8. Change points of SGE after a scale in

to detect changes in the same metrics. Unfortunately, we cannot provide yet a quantitative measure on the accuracy of the delay estimation because the systems under investigation lacked the necessary monitoring features to measure it. Hence the question about which algorithm is better is still open to debate.

In general, the algorithms that we considered detected more points than we expected. Nevertheless, by exploiting the action-effect classification, the knowledge about the time of reconfiguration triggering, and the correlation among different metrics our framework was able to filter out the spurious changes. Thus we conclude that, at least in elastic systems that we considered, our framework was able to properly estimate the actuation delays.

During this study we gained experience in the use CPD algorithms that we summarize here in the form of guidelines. Obtaining good results with change point algorithms is a matter of choosing how to configure and when to apply them, i.e., under what conditions and on which system metrics. The two algorithms that we considered in this work are different both in how they operate and how they can be configured:

1) PELT can be used to find changes in mean, variance or both. We noticed that for monotonically increasing or decreasing signals, finding changes in the variance can be a slightly better choice than the mean if signals do not change too much. Otherwise PELT may show a behavior similar to the one presented in the SGE case study. We also noticed that finding changes in the mean is a good choice when signals vary with high amplitude variations and frequent spikes.

PELT has one main configuration parameter, namely the *penalty*, that guards the algorithm against over-fitting, that is, finding too many change points. In the current implementation, this parameter may take several values but we choose to set it manually. For Cassandra, we have found that the penalty should be set between 0.1-1 when PELT is applied to the variance, while it should be set the maximum value of the data series when it is applied to the mean. For the SGE case we found its value empirically by comparing results across different experimental trials.

2) BOCPD finds changes by leveraging a model that predicts the expected behavior of the signals, therefore it is not tied to any statistical moment. We noticed that BOCPD was able to detect any of the inspected change but in some cases, especially if the signals present sharp and frequent variations, may detect too many changes. This happened when the algorithm could not retrain its internal model fast enough to keep the pace of signal variations. For this cases we suggest to switch to PELT.

BOCPD has one main configuration parameter, namely the *size* of training set. This parameters govern the amount of observations that the algorithm uses to train the first time its internal model. We set the size of the training to a small value, between zero and 10% of the observations, to force the algorithm to adapt to the current signal as it evolves. With this settings we noticed that the number of change points detected were the closest to the expected one. Furthermore, by using bigger training sets there is the risk to train the model using data before and after the change. When this happened the algorithm was not be able to distinguish anymore between the situation before and after the change, completely failing to detect the right changes.

V. Related Work

Controlling elastic computing systems has gained importance with technologies promoting adaptivity like cloud computing and elastic storage. Konstantinou et al. [17] describe TIRAMOLA, a framework for monitoring and resizing NoSQL clusters. Controlling elastic storage is also approached by Lim et al. [20] that describe actuator delays, in this case due to rebalancing, as one of the main challenges. Lim and co-authors propose different rebalancing strategies for reducing the delays. In contrast with this, we propose a generic approach to measure actuation delays that is not limited to the case of elastic storage. Another contribution towards actuation delay monitoring in cloud computing is described by Ming et al. [23] that measure the start-up time of virtual machines on different cloud IaaS providers. They emphasize that the time of boot the VMs that is shown by the provider differs from the actual time to start the operating system. Similarly, our work for detecting actuation delays captures the time that is needed for all the software on the virtual machine to start, but it also includes the time need before serving user requests.

Action modeling is the key in developing automatic controllers, and without knowing at least the approximate effects of an action on the system, it is impossible to plan a suitable control policy for the elastic system. Considering a series of observations, action modeling outputs the type of action that could lead to these observations, with its main characteristics [1], [26]. Bodik et al. [3] propose learning action effects through the use of statistical models in the context of data centers. The models are used to predict the effect of control

action on performance. Estimating action effects plays a considerable role also in areas such as planning, scheduling and design of automatic controllers. Even though the time needed for the adaptation actions to complete is emphasized as being important [13], and algorithms considering this information in their planning process were proposed [15], to the author knowledge there is no approach that attempts to estimate the actuation delays. This work proposes a framework for control action delay estimation using change point detection techniques, highlighting the delay estimation challenges and validating this framework on real world scenarios.

VI. Conclusions and Future Work

Identifying actuation delays in cloud based elastic systems is crucial for the design of controllers that provide acceptable performance. In this paper, we analyzed in detail issues, such as action-effect classes and requirements, that relate to the estimation of actuation delays. Based on that, we proposed a novel framework that leverages state-of-the-art change point detection algorithms to address the actuation delay estimation. We have provided a prototype implementation and illustrated the usefulness of our framework with two real-world elastic systems in the cloud.

In our future work, we will conduct experiments with other types of cloud systems to examine the accuracy of our detection framework for other action-effect classes. We are also working on steering the monitoring in order to improve the accuracy of actuation delay detection.

Acknowledgments

This work was partially supported by the Swiss National Science Foundation under the "Fellowship for Prospective Researches" contract PBTIP2-142337 and by the European Commission in terms of the CELAR FP7 project (FP7-ICT-2011-8 #317790).

References

[1] E. Amir and A. Chang. Learning Partially Observable Deterministic Action Models. *Journal of Artificial Intelligence Research*, 33:349–402, 2008.

[2] M. Basseville and I. V. Nikiforov. *Detection of abrupt changes: theory and application*. Prentice-Hall, Inc., 1993.

[3] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Workshop on Hot topics in cloud computing*, HotCloud'09. USENIX Association, 2009.

[4] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer-Verlag, 2009.

[5] S. Clayman, A. Galis, C. Chapman, G. Toffetti, L. Rodero-Merino, L. M. Vaquero, K. Nagin, and B. Rochwerger. Monitoring Service Clouds in the Future Internet. In *Towards the Future Internet - Emerging Trends from European Research*. IOS Press, 2010.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *ACM symposium on Cloud computing*, SoCC'10, pages 143–154, 2010.

[7] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar. SYBL: an Extensible Language for Controlling Elasticity in Cloud Applications. In *IEEE International Symposium on Cluster Computing and the Grid (to appear)*, CCGRID'13, 2013.

[8] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck. From data center resource allocation to control theory and back. In *IEEE International Conference on Cloud Computing*, CLOUD'10, pages 410–417, 2010.

[9] I. Epifani, C. Ghezzi, and G. Tamburrelli. Change-point detection for black-box services. In *International symposium on Foundations of Software engineering*, FSE'10, pages 227–236, 2010.

[10] S. Fritsch, A. Senart, D. C. Schmidt, and S. Clarke. Scheduling time-bounded dynamic software adaptation. In *International workshop on Software Engineering for Adaptive and self-Managing Systems*, SEAMS'08, pages 89–96, 2008.

[11] S. Fritsch, A. Senart, D. C. Schmidt, and S. Clarke. Time-bounded adaptation for automotive system software. In *International Conference on Software Engineering*, ICSE'08, pages 571–580, 2008.

[12] A. Gambi, G. Toffetti Carughi, C. Pautasso, and M. Pezzè. Kriging controllers for cloud applications. *IEEE Internet Computing*, accepted for publication, 2012.

[13] B. Jackson, J. Scargle, D. Barnes, S. Arabhi, A. Alt, P. Gioumousis, E. Gwin, P. Sangtrakulcharoen, L. Tan, and T. T. Tsai. An algorithm for optimal partitioning of data on an interval. *IEEE Signal Processing Letters*, 12(2):105 – 108, 2005.

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

[15] M. A. Khan, D. Turgut, and L. Boloni. Optimizing coalition formation for tasks with dynamically evolving rewards and nondeterministic action effects. *Autonomous Agents and Multi-Agent Systems*, 22:415–438, 2011.

[16] R. Killick, P. Fearnhead, and I. A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598, 2012.

[17] I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas. Tiramola: elastic nosql provisioning through a cloud management platform. In *International Conference on Management of Data*, SIGMOD'12, pages 725–728, 2012.

[18] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *ACM SIGCOMM conference on Internet Measurement*, IMC'10, pages 1–14, 2010.

[19] J. Z. Li, J. W. Chinneck, C. M. Woodside, and M. Litoiu. Fast scalable optimization to configure service systems having cost and quality of service constraints. In *International Conference on Autonomic Computing*, ICAC'09, pages 159–168, 2009.

[20] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *International Conference on Autonomic Computing*, ICAC'10, pages 1–10, 2010.

[21] L. Ljung, editor. *System identification (2nd ed.): theory for the user*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[22] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *International Conference on Autonomic computing*, ICAC'11, pages 131–140, 2011.

[23] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *IEEE International Conference on Cloud Computing*, CLOUD'12, pages 423–430, 2012.

[24] A. J. Ramirez, D. B. Knoester, B. H. Cheng, and P. K. Mckinley. Plato: a genetic algorithm approach to run-time reconfiguration in autonomic computing systems. *Cluster Computing*, 14:229–244, 2011.

[25] J. Reeves, J. Chen, X. L. Wang, R. Lund, and Q. Q. Lu. A review and comparison of changepoint detection techniques for climate data. *Journal of Applied Meteorology and Climatology*, 46(6):900–915, 2007.

[26] D. Shahaf, A. Chang, and E. Amir. Learning Partially Observable Action Models: Efficient Algorithms. In *National Conference on Artificial Intelligence*, 2006.

[27] A. Soule, K. Salamatian, and N. Taft. Combining filtering and statistical methods for anomaly detection. In *ACM SIGCOMM conference on Internet Measurement*, IMC'05, pages 31–31, 2005.

[28] G. Toffetti, A. Gambi, C. Pautasso, and M. Pezzè. Engineering autonomic controllers for virtualized web applications. In *International Conference on Web Engineering*, ICWE'10, 2010.

[29] R. Turner, Y. Saatçi, and C. E. Rasmussen. Adaptive sequential Bayesian change point detection. In Z. Harchaoui, editor, *Temporal Segmentation Workshop at NIPS 2009*, Whistler, BC, Canada, 2009.

[30] H. Wang, D. Zhang, and K. Shin. Change-point monitoring for the detection of DoS attacks. *IEEE Transactions on Dependable and Secure Computing*, 1(4):193–208, 2004.