Technological University Dublin
ARROW@TU Dublin

Conference papers | School of Computer Sciences

2019

A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster

Basel Magableh

Muder Almiani

Follow this and additional works at: https://arrow.tudublin.ie/scschcomcon

 Part of the Computer Sciences Commons

This Conference Paper is brought to you for free and open access by the School of Computer Sciences at ARROW@TU Dublin. It has been accepted for inclusion in Conference papers by an authorized administrator of ARROW@TU Dublin. For more information, please contact arrow.admin@tudublin.ie, aisling.coyne@tudublin.ie, gerard.connolly@tudublin.ie.

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 4.0 License

ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/331790124

A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster

Chapter · January 2020

DOI: 10.1007/978-3-030-15032-7_71

CITATIONS
3

READS
2,372

2 authors:

Basel Magableh
Technological University Dublin - City Campus
26 PUBLICATIONS **72** CITATIONS
SEE PROFILE

Muder Almi'ani
Gulf University for Science and Technology (Kuwait)
57 PUBLICATIONS **234** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Intelligent Intrusion Detection Systems View project

Project Deep Recurrent Q-Network View project

All content following this page was uploaded by Basel Magableh on 09 April 2019.

The user has requested enhancement of the downloaded file.

A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster

Basel Magableh[1(✉)] and Muder Almiani[2]

[1] School of Computer Science, Technological University Dublin, Dublin, Ireland
basel.magableh@dit.ie
[2] Al-Hussein Bin Talal University, Ma'an, Jordan
malmiani@my.bridgeport.edu

Abstract. One desired aspect of a self-adapting microservices architecture is the ability to continuously monitor the operational environment, detect and observe anomalous behaviour as well as implement a reasonable policy for self-scaling, self-healing, and self-tuning the computational resources in order to dynamically respond to a sudden change in its operational environment. Often the behaviour of a microservices architecture continuously changes over time and the identification of both normal and abnormal behaviours of running services becomes a challenging task. This paper proposes a self-healing Microservice architecture that continuously monitors the operational environment, detects and observes anomalous behaviours, and provides a reasonable adaptation policy using a multi-dimensional utility-based model. This model preserves the cluster state and prevents multiple actions to taking place at the same time. It also guarantees that the executed adaptation action fits the current execution context and achieves the adaptation goals. The results show the ability of this model to dynamically scale the architecture horizontally or vertically in response to the context changes.

Keywords: Self healing · Microservices architecture · Anomaly detection · Run-time configuration

1 Introduction

A microservices architecture could be defined in the context of a service-oriented architecture as a composition of tiny fine-grained distributed loosely coupled building blocks of software components [27]. In a microservices cluster, the performance of its nodes might fluctuate around demands to accommodate scalability, orchestration and load balancing issued by the leader node. To achieve an optimal level of performance, the architecture requires a model that (i) is able to detect anomalies in real-time, (ii) with a high accuracy and (iii) leads to a low rate of false alarms. In addition, a set of possible configurations should be designed and incorporated in the architecture in order to adapt itself to the

© Springer Nature Switzerland AG 2020
L. Barolli et al. (Eds.): AINA 2019, AISC 926, pp. 846–858, 2020.
https://doi.org/10.1007/978-3-030-15032-7_71

changes in its operational environment. This adaptability requires a dynamic decision making component that is capable of selecting a desired cluster state according to a set of constraints. This research proposes a method to continuously observe and monitor the Docker Swarm cluster state and detect anomalous behaviour. This method also aims at equipping a microservices architecture with adaptation strategies able to reason about detected anomalies detected able to self-adjust its parameters and to verify its actions at runtime without human intervention. In details, the proposed method offers microservices architecture a self-adaptation property by following the MAPE-K (Monitor-Analyse-Plan-Execute over a shared Knowledge) model. The main contribution of this work is the employment of a utility function in the process of adaptation.

The remainder of the paper is structured as follows: Sect. 2 provides an overview of self-healing architectures and surveys the approaches for anomaly detection and run-time configuration. Section 3 presents a model that can continuously observe microservices architecture with self-healing capabilities. Adaptation planning and execution is discussed in Sect. 3.2. The implementation of this model is discussed in Sect. 3.3. Section 3.4 is focused on presenting results followed by a critical discussion of the effectiveness of this model. Section 4 summarises this research, highlighting its contribution and setting future work.

2 Related Work

A microservices architecture is a composition of tiny fine-grained distributed loosely coupled building blocks of software components [27]. A self-healing architecture refers to the capability of its software components to discover, diagnose and react to disruptions. Such architecture can also anticipate potential problems and, accordingly, take suitable actions to prevent a failure by self-adaptation [14]. In order to achieve this, a microservices architecture require a decision-making strategy that can work in real-time and is able to reason about its own state and its surrounding environment in a closed control loop and then to act accordingly [4]. Typically, a self-adapting architecture should implement the MAPE-K (Monitor-Analyse-Plan-Execute over a shared Knowledge) approach. This approach includes: (i) gathering of data related to the surrounding context (context sensing); (ii) context observation and detection; (iii) dynamic decision making; (iv) execution of adaptation through actions to achieve a set of objectives defined as QoS; (v) verification/validation of the applied adaptation actions in terms of accuracy in meeting the adaptation objectives.

A number of approaches exist for achieving high degrees of self-adaptability. For instance, in [26], self-adaptability involves context sensing and collection, observation and detection of contextual changes in an operational environment. Self-adaptation in an architecture and a dynamic adjustment of its behaviour can be achieved using parameter-tuning [5], component-based composition [18], or middleware-based approaches [6]. An important aspect of a self-adaptive system is related to its ability to validate and verify the adaptation action at run-time. This can be done by employing game theory [28], utility theory [15] or a model-driven approach as in [24]. Context information refers to any information that

is computationally accessible and upon which behavioural variations depend [13]. Context observation and detection approaches are used to detect abnormal behaviour within the microservices architecture at run-time. Related work in context modelling, context detection and engineering self-adaptive software systems are discussed in [4,8,23,26]. In dynamic decision making and context reasoning an architecture should be able to monitor and detect normal/abnormal behaviour by continuously monitoring the contextual information found in the microservices cluster. There are two phases for detecting anomalies in a software system: a training phase which involves profiling the normal behaviour of the system; and a phase aimed at testing the learned profile of the system with new data and employing it to detect normal/abnormal behaviours [19].

Three major techniques for anomaly detection have emerged from the literature: (a) statistical anomaly detection, (b) data-mining and (c) machine learning-based techniques. In the statistical methods, the anomaly detection algorithm usually observes the activity of the software system and generates profiles with system metrics such as CPU and memory to represent its behaviour. Various statistical anomaly detection systems have been proposed as in [2,22]. They provide accurate notifications of malicious attacks that occur over long periods of time and this class of systems performs better than the other classes in detecting denial-of-service attacks [19]. In statistical anomaly detection a skilled attacker might train a statistical anomaly detection system to accept the abnormal behaviour as normal. It is difficult to determine the thresholds that make a balance between the likelihood of a false negative – the system fails to identify an activity as an abnormal behaviour – and the likelihood of a false positive (false alarms). Therefore, statistical anomaly detection systems need an accurate model with precise distributions for each metric. In practice, the behaviour of virtual machines/computers cannot be entirely modelled using solely statistical methods. Data-mining anomaly detection techniques are about finding insights which are statistically reliable, previously unknown, and actionable from data [21]. The traditional data-mining process involves discovering a novel, distinguished and useful data pattern in large datasets to extract hidden relationships and information. However, two issues exist in anomaly detection in microservices architectures: a lack of a large dataset containing information about the architecture itself, and the small number of approaches applied to these architectures [21]. Machine learning-based anomaly detection models are data-driven and are mainly focused on learning exclusively from past data [19]. When additional and new data becomes available, they can intrinsically influence the detection strategy and classify significant deviations from the normal behaviour of an underlying software programme. Therefore, they need to be often retrained to be in line with current data. These approaches generally use a combination of clustering and classification algorithms to detect anomalies. The former are used to cluster the dataset and label observations. The latter algorithms, such as decision trees can then be used for classification to distinguish between normal/abnormal behaviour [3]. Other applications and information can be found in [3,10,11]. It is important to highlight that, due to the opening deployment

and limited resources generally found in a microservices cluster, a lightweight approach to data clustering/classification should be used.

Other anomaly detection approaches exist in the literature. For example, the Numenta Platform for Intelligent Computing (NUPIC) is based on the hierarchical temporal memory (HTM) model proposed in [12]. This has been experimentally applied in real-time anomaly detection of streaming data [17] and it is claimed to be efficient and tolerant to noisy data, capable to adapt to the changes of data statistics. It can also detect extremely subtle anomalies with a very minimal rate of false positives. In a similar study, Ahmad et al. [1] proposed an updated version of this anomaly detection algorithm introducing the anomaly likelihood concept. The anomaly score calculated by the NUPIC algorithm represents an immediate calculation of the predictability of the current input stream. This approach works very well with predictable scenarios in many practical applications. As there is no noisy and unpredictable data found, the raw anomaly score gives an accurate prediction of false negatives. However, the changes in predictions would lead to revealing anomalies in the system's behaviour. Thus, instead of using the raw anomaly score, authors in [1] proposed a method for calculating the anomaly likelihood by modelling the distribution of anomaly scores and by using it to check the likelihood of the current state of the system to identify anomalous behaviour. The anomaly likelihood is metric which defines how anomalous the current state is based on the prediction history calculated by the HTM model. The anomaly likelihood is calculated by maintaining a window of the last raw anomaly scores and then calculating the normal distribution over the last obtained/trained values. The most recent average of anomalies is then calculated using the Gaussian tail probability function (Q-function) [7].

3 Design and Methodology

This research focuses on proposing a mechanism that can continuously observe and monitor the microservices architecture and be able to detect anomalous behaviour with high accuracy and generate low rate of false alarms. At the same time, this mechanism should be able to respond to true positive alarms by suggesting a set of adaptation policies (adaptation strategy), that can be deployed in the cluster to achieve high level of self-healing in response to changes in its operating environment. The envisioned property of this mechanism is that it can be easily deployed with fewer and smaller footprints on the limited resources found in the tiny containers running in a microservices cluster.

3.1 Self-healing Microservices Architecture

One important aspect of a self-healing microservices architecture is the ability to continuously monitor the operational environment, detect and observe anomalous behaviour, and provide a reasonable policy for self-scaling, self-healing, and self-tuning the computational resources to adapt a sudden changes in its operational environment dynamically at run-time. A typical microservices architecture

is shown in Fig. 1 and it was designed according to the MAPE-K model (Monitor-Analyze-Plan-Execute over a shared Knowledge) [25].

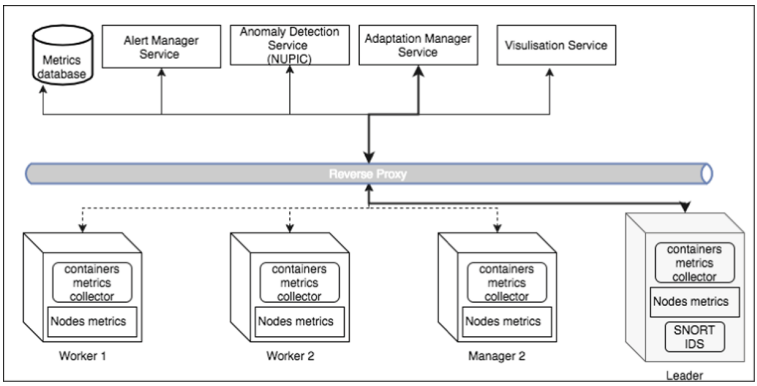

Fig. 1. A microservices architecture implemented in Docker Swarm [25].

This model offer to the microservices architecture the following functionalities:

- **Metric collection:** continuous collection of fine-grained metrics about cluster nodes, services and containers such as CPU usage, Memory, Disk Reads Bytes/sec, Network Read/s, network write/s and Disk Writes Bytes/sec) which is streamed into the anomaly detection service at real-time;
- **Model Training:** the NUPIC anomaly detection service [1] continuously runs over the streamed metrics stored in a database, enabling the training of a model with the collected metrics;
- **Anomaly Detection:** collected real-time data is feed on the fly to the NUPIC anomaly detection service, which provides two features: continuous detection of anomalous behaviour with high accuracy and predictions about the architecture performance based on historic data. This service can alert the architecture about incoming spike on resources demand which can then be used by the adaptation manager to schedule a proactive adaptation strategy ahead of time. In addition, it is able to detect anomalies as early as possible before the anomalous behaviour interrupts the functionality of the running services in the cluster Ahmad et al. [1];
- **Adaptation Election:** once an anomalous behaviour is detected, anomaly score and likelihood are calculated by the Anomaly Detection Service as in Fig. 1. The alert manager services notifies the adaptation manager about the anomaly detected and then selects the adaptation action(s) after calculating the utility value for each of the possible actions, as detailed in Sect. 3.2. Subsequently, the Adaptation Manager uses the input of the anomaly likelihood, architecture constraints (specified by the DevOp during deployment)

and desired/predicted QoS to calculate the best variation of the adaptation that has the highest utility;

- **Adaptation Execution:** the adaptation manager executes the strategies according to the aggregated value of the utility returned by the algorithm. Once the adaptation action is completed, a set of adaptation actions are deployed in the architecture. To avoid, conflicts between multiple adaptation polices, the adapter allows the adaptation actions to be fully completed and verified by the cluster leader according to the consensus performed by the RAFT algorithm [20]. It then set a cool off timer before initiating new adaptation actions. This technique is used to avoid resources thrashing and preserving the cluster state for auto-recovery. The adaptation manager then sends to the cluster leader a set of instructions that might involve tuning of cluster parameters – horizontal scaling – adding/removing nodes or vertical scaling of microservice's containers like scaling a service in/out;
- **Adaptation Verification:** The cluster leader and all managers in the cluster subsequently vote on the adaptation action based on the RAFT consensus algorithm [20]. The results of the vote are used to validate and verify the adaptation action. If the adaptation action passes the voting process, it will be executed by the cluster leader and the adaptation manager records the adaptation attempt as successful. Otherwise, the adaptation manager keeps the current state of the cluster and records the adaptation attempt as failed. In both cases, the adaptation manager records the number of attempts used to complete the adaptation actions.

3.2 Adaptation Election

To provide the model, described in the previous section, with dynamic policy election that guarantees high accuracy of selecting the best adaptation action that fits in the current execution context, an extension of the adaptation manager with a policy election process by employing a utility function is proposed. This function is aimed at computing the probability of transition from one state to another. In this process the anomaly detection service plays a significant role. At each state s of the microservices architecture, there is a set of context values CV: $c_1, \ldots, c_m$ measuring the metrics in the operating environment such as CPU, Memory, Disk I/O and Network. The anomaly detection service reads the current values of all metrics in CV and NUPIC calculates the anomaly scores AS: $as_1, \ldots, as_m$ and anomaly likelihoods AL: $al_1, \ldots, al_m$ in the current state for each of them. The anomaly likelihood accurately defines how anomalous the current metric is when compared to the distribution of the values learned by the anomaly detection service. The anomaly score and the anomaly likelihood are scalar values in [0..1]. For instance, if the al_{cpu} is 1 and c_2 is the CPU value of 70%, then c_2 can be associated with a high utility score and it might be considered in the next adaptation action. In turn, the adaptation manager can select an adaptation policy able to reason about the anomalous behaviour of the CPU. In another scenario, if the anomaly likelihood is 0, then the metric can be

associated with a low utility score so it will not be considered in the subsequent adaptation action.

$$W(al_m, C_m) = \sum_{i=1}^{m} al_i \cdot c_i \tag{1}$$

$$U(cv_x) = \max(\sum_{t=1}^{20} cv_x^i \times al_x) \tag{2}$$

in Eq. 2 CV the vector contains the context value *cv*s of the context metric, AL the vector containing all the anomaly likelihood *al* for each metric returned by NUPIC. These are taken 20 times i in a time window that is set to 300 s.

From utility theory, a von Neumann-Morgenstern utility function $U_i : X_i \rightarrow \mathbb{R}$ assigns a real number to each quality dimension i, which we can normalize to the range [0, 1] [9]. Across multiple dimensions of contextual changes C_m, we can attribute a percentage weight to each dimension to account for its relative importance compared to other dimensions. These weights form the utility preferences. The overall utility is then given by the utility preference $w(c_m)$ function calculated using Eq. 1. For example, if three objectives, u(cpu), u(memory), u(disk), are given anomaly likelihood as follows: the first is twice as important as the second, and the second is three times as important as the third. Then the weight would be quantified as $[w1 : 0.6, w2 : 0.3, w3 : 0.1]$, where the weight is the Anomaly Likelihood of each metric. This gives the CPU metric higher priority to be consider in the adaptation action.

In this paper. We argue that the use of anomaly likelihood to weight the collected metrics provides an accurate calculation of the utility function and provides the model with better estimation of the adaptation action. So the maximum metric is selected using the equation described in 2, which select the maximum W of specific metric value that has the highest Anomaly Score returned by the Anomaly detection service.

$$Cost(u_m) = \frac{(Current(c_m) - NUPICPredicted(c_m)) \cdot (as_m) \cdot al_m, c_m))}{UsageTime * InstanceMonthlyPrice} \tag{3}$$

where $Current(c_m)$ is the current value of the metric C_M, $NUPICPredicted$ (c_m) is the predicted value of C_M computer by the NUPIC algorithm, $AnomScore(c_m)$ is the anomaly score of C_m at time t_i calculated by NUPIC, and W_i is the anomaly likelihood for metric C_M as per Eq. 1. The $UsageTime$) refers to the total number of hours the node is expected to be used per day (constant value). The $InstanceMonthlyPrice$ is the price in $ (dollars) for provisioning an instance per month. Normally this is a constant price specified by the cloud infrastructure provider based on the instance type.

$$change_m = (W_a(al_m, C_m) - W_b(al_m, C_m)) \tag{4}$$

where i is the current time stamp, m is a future time stamp
we calculate the weight of the current metric C_m and the previous value.

3.3 Experimental Setup and Evaluation Strategy

To validate the ideas presented in this paper, we design and develop a working prototype of Microservice architecture in Docker swarm[1] as shown in Fig. 1. The cluster consisted of manager and worker nodes. Each cluster has one leader, which maintains the cluster state and preserves the cluster logs. Also, the leader node initializes the vote of Raft Consensus Algorithm [20] to agree/disagree on specific value based on the consensus by all nodes in the cluster. Only the leader node is allowed to commit and save the variable values or logs. To meet scalability and availability, the leader node distributed the work load between the workers based on Raft Consensus Algorithm [20]. This means that each service could be executed by assigning multiple containers across the cluster's nodes.

The main services implemented in this architecture are: Time series metrics database for context collection, Nodes metrics used to collect metrics from all nodes in the cluster, Alert and notification manager used to notify the adaptation manager about contextual changes offered by Prometheus framework[2]. Docker containers metrics collector for collecting fine-grained metrics about all running containers in all nodes[3]. Reverse proxy for routing traffic between all services in the cluster[4]. Unsupervised Real-time Anomaly Detection based on NUPIC[5], Adaptation manager for executing, validating the adaptation actions developed as a prototype of this research. Time series analytic and visualisation dashboard for observing the behaviour of the Microservices cluster[6].

This live snapshot[7] provides a full virtualisation of all services running in the cluster. The evaluation of the effectiveness of this model will be based on calculating a utility function for all metrics monitored, then it will calculate the number of adaptation attempts, successful convergence of services/nodes, or errors which leads to unstable state of the cluster.

The evaluation of the model is set over two stages: (i) assessing the consistency of the behaviour of the cluster by evaluating the state of the swarm after a sequence of adaptation actions. The idea is to start with no nodes and the utility model should be able to create a new cluster and add the required number of nodes/replicas until it reaches an stable state. In other words, this when the cluster reaches a convergence and all the services in it are accessible and available. The decision is then left for the adaptation manager to scale the cluster horizontally or vertically until it reaches the sable state. (ii) evaluating the accuracy of the model in electing the correct adaptation action by identifying the highest metric value that need to be consider in the adaptation and the selected adaptation action. In summary the evaluation criteria are: (a) **criterion**

[1] https://docs.docker.com/engine/swarm/.
[2] https://prometheus.io.
[3] https://github.com/google/cadvisor.
[4] https://caddyserver.com/docs/proxy.
[5] http://nupic.docs.numenta.org/stable/index.html.
[6] https://grafana.com.
[7] https://snapshot.raintank.io/dashboard/snapshot/UJlrTzwubrRQjDwM1YFJle5zvdK3Anr7?orgId=2.

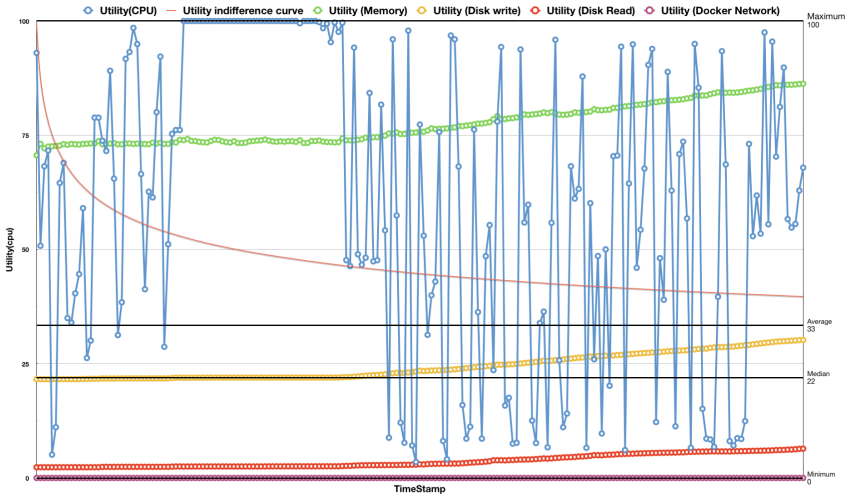

Fig. 2. Dimensional analysis of variations of utility functions

1: the ability of the model to manage a microservices cluster and to scale it horizontally or vertically until it reaches an stable state (when all the services are available). (b) **criterion 2:** the ability of the model to handle dynamic changes in the cluster and to dynamically adapt to sudden changes, such as simulated stress test or Distributed Denial of Service attack (DDOS), without leading the cluster to an unstable state; (c) **criterion 3:** the ability of the architecture to meet demands dynamically and maintain a stable state for the cluster.

3.4 Discussion

To test the first evaluation criteria, a stress test has been executed in the cluster manager until its CPU usage reaches 70%, which triggers an alert to the adaptation manager. In turn, the adaptation manager collects the current reading of the metrics, anomaly score and anomaly likelihood produced by NUPIC. Then, it calculates the rate of changes for each metric in order to elect the metric that has the highest utility. As example, Fig. 2 depicts the utility of CPU usage, memory usage, disk reads (bytes/s), disk writes (bytes/s), docker network (sent/received bytes/s). The CPU usage has the highest utility as confirmed by the utility indifference indicator (curve in Fig. 2). Also, the memory usage of the service shows slow rate of changes over time, which make it optional to be considered in the adaptation action. With regard to the utility of disk read/write, there is no divergence above the moving average (utility indifference curve) so it is not considered in the subsequent adaptation action. The docker network shows no changes over time as the load balancer and the reverse proxy manage to divert the traffic to many containers distributed in the cluster. As the $U(CPU)$ has the highest value of changes, this triggers an adaptation action and it allows the model to reason about the high demand of CPU usage. As a consequence, the

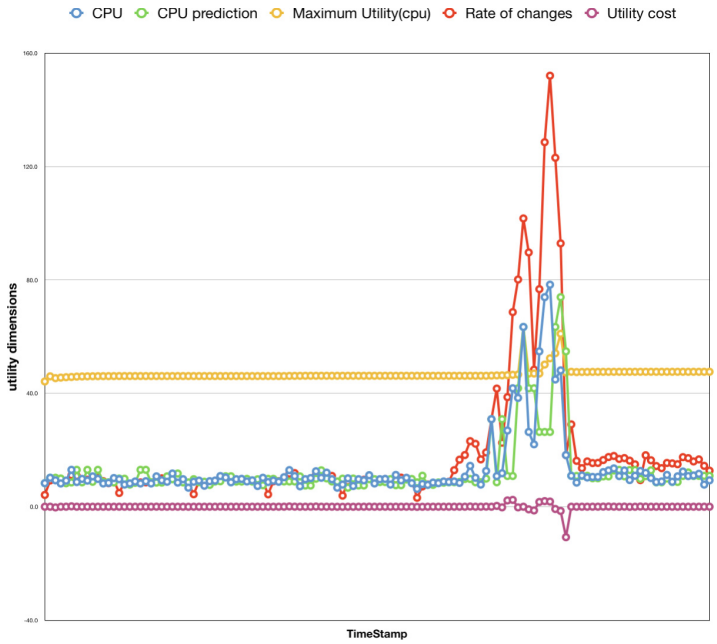

Fig. 3. Utility(cpu) rate of changes and cost calculated based in Eqs. 3 and 4

adaptation manager creates additional nodes and add them to the swarm cluster automatically. The number of nodes is equal to the utility calculated as in Eq. 2. This lead to new nodes addition to the swarm as shown in the snapshot[8] (a full visualised and analytics dashboard of the swarm after the adaptation). Once the CPU demand is reduced, the adaptation manager calculates the variations of the utility and remove a number of nodes equal to the value returned by the cost function of Eq. 3. A snapshot[9] of the system after executing the adaptation action to reason about the low level of the CPU usage. This satisfies the evaluation criteria 2.

The accuracy of the utility cost, rate of changes, and the maximum utility dimension are vital for the success of the adaptation process. So, Fig. 3 depicts the calculation of the rate of changes and the utility cost to reach the desired number of nodes/replicas needed. The calculation accurately satisfies the adaptation objectives and provides the architecture with a suitable number of required nodes/replicas. As shown in Fig. 3, this number increases at the right time when the CPU demand spikes. In fact, the number of nodes/replicas reduces just before the CPU demand declines significantly. The rate of changes in CPU usage

[8] https://snapshot.raintank.io/dashboard/snapshot/sstuT2tuYkob8zjIbh1YXzBYxSJDFd9z?orgId=2.

[9] https://snapshot.raintank.io/dashboard/snapshot/UJlrTzwubrRQjDwM1YFJle5zvdK3Anr7?orgId=2.

declined so the utility function returns a negative value for the required number of nodes/replicas as long they are above the minimum amount specified by the Dev-Ops. Also, as shown in Fig. 3 the utility cost normalizes and tunes the CPU demand. This provides evidence that the adoption of the utility provides the adaptation cycle with a dynamic variability over the needed/allocated resources rather than scaling the architecture in/out according to a static threshold. This satisfies the evaluation criteria 3.

In another scenario, a Distributed Denial of Service attack to a web service running in the swarm was simulated. This was aimed at verifying that the adaptation manager can accommodate the DDOS attack by adding more replicas to the service. As in the proposed model, if the anomaly detection service would consider a specific record as anomalous and this was an actual anomaly, then this attempt is classified as a **True Positive**. If the anomaly detection service consider the data as normal behaviour, and it is actually normal data, then this attempt is classified as **True Negative**. If it classifies an anomalous behaviour as normal behaviour, then it means that NUPIC fails to detect the anomaly and this attempt is classified as **False Negative**. Eventually, if the service classifies the data as anomalous behaviour but the data actually corresponds to a normal behaviour, then this attempt is considered a **False Positive** (false alarm). Both True Positive and False Positive are important benchmarks to measure the accuracy of the intrusion detection mechanism. Table 1 summarizes the confusion matrix emerged during our experiment and it provides a better pictures of the accuracy of the anomaly detection algorithm. The confusion matrix will be used to calculate the model detection rate, false positive, and accuracy, which achieves the second objective of the evaluation (criterion 3).

Table 1. Results of the proposed anomalies detection model on confusion matrix

X = 1528	Predicted anomalies	Predicted normality
Actual anomalies (TP + FN) (55)	TP = 49	FN = 6
Actual normalies (FP + TN) (1473)	FP/ False Alarm = 38	TN = 1435

The true positive rate (TPR), sensitivity or recall is $49/(49 + 6) = 89\%$, the false positive rate (FPR) is $38/1473 = 2.5\%$, the false negative rate (FNR) is $6/(49 + 6) = 11\%$ and the true negative rate (TNR), specificity or selectivity is $1435/1473 = 97.5\%$. The precision (PPV) of the algorithm is $49/(49 + 38) = 56.3\%$ while its false omission rate (FOR) is $6/6 + 1435 = 4.4\%$. The false discovery rate (FDR) is $38/49 + 38 = 43.7\%$ while its negative predicted value (NPV) is $1435/6 + 1435 = 99.5\%$. Thus the overall accuracy of the algorithm is $(49 + 1435)/(49 + 38 + 6 + 1435) = 97.1\%$. Since the actual tests is unbalanced, having more normalies than anomalies, then the balanced accuracy is computed: $(0.89 + 0.975)/2 = 93.25\%$. Overall, the model has a low precision score (PPV) and an high recall score (TPR). This means that the model has a moderate degree of exactness but a high completeness.

4 Conclusions and Future Work

This model manages to offer the Microservices architecture with continuous monitoring, continuous detection of anomalous behaviour, and provides the architecture with dynamic decision making based on the employment of multidimensional utility-based model. The results in above, shows high accuracy in detecting the anomaly and an accurate calculation of variant adaptation actions. It Also shows high success rate in performing horizontal and vertical scaling in response to contextual changes. The uses of utility-based model enables the architecture to dynamically elect a reasoning approach based on the highest utility dimension of context changes in the operational environment.

References

1. Ahmad, S., Lavin, A., Purdy, S., Agha, Z.: Unsupervised real-time anomaly detection for streaming data. Neurocomputing **262**(Suppl. C), 134–147 (2017)
2. Anderson, D., Frivold, T., Valdes, A.: Next-generation intrusion detection expert system (NIDES): a summary. SRI International, Computer Science Laboratory Menio Park, CA (1995)
3. Buczak, A.L., Guven, E.: A survey of data mining and machine learning methods for cyber security intrusion detection. IEEE Commun. Surv. Tutor. **18**(2), 1153–1176 (2016)
4. Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Malek, R.M., Müller, H., Park, S., Shaw, M., Tichy, M.: Software engineering for self-adaptive systems: a research road map (draft version). In: Dagstuhl Seminar Proceedings 08031 (2008)
5. Cheng, S.W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the rainbow self-adaptive system. In: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009, pp. 132–141. IEEE (2009)
6. Cheung-Foo-Wo, D., Tigli, J.Y., Lavirotte, S., Riveill, M.: Self-adaptation of event-driven component-oriented middleware using aspects of assembly. In: Proceedings of the 5th International Workshop on Middleware for Pervasive and Ad-Hoc Computing: Held at the ACM/IFIP/USENIX 8th International Middleware Conference, pp. 31–36 (2007)
7. Craig, J.W.: A new, simple and exact result for calculating the probability of error for two-dimensional signal constellations. In: Conference Record, Military Communications in a Changing World, Military Communications Conference, MILCOM 1991, pp. 571–575. IEEE (1991)
8. De Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: Software Engineering for Self-Adaptive Systems II, pp. 1–32. Springer (2013)
9. Fishburn, P.C., Kochenberger, G.A.: Two-piece von Neumann-Morgenstern utility functions. Decis. Sci. **10**(4), 503–518 (1979)
10. Golmah, V.: An efficient hybrid intrusion detection system based on C5.0 and SVM. Int. J. Database Theory Appl. **7**(2), 59–70 (2014)
11. Haq, N.F., Onik, A.R., Hridoy, M.A.K., Rafni, M., Shah, F.M., Farid, D.M.: Application of machine learning approaches in intrusion detection system: a survey. IJARAI-Int. J. Adv. Res. Artif. Intell. **4**(3), 9–18 (2015)

12. Hawkins, J., Blakeslee, S.: On Intelligence. Macmillan, London (2007)
13. Hirschfeld, R., Costanza, P., Nierstrasz, O.M.: Context-oriented programming. J. Object Technol. **7**(3), 125–151 (2008)
14. Horn, P.: Autonomic computing: IBM's perspective on the state of information technology. Technical report (2001)
15. Kakousis, K., Paspallis, N., Papadopoulos, G.A.: Optimizing the utility function-based self-adaptive behavior of context-aware systems using user feedback. In: OTM Confederated International Conferences "On the Move to Meaningful Internet Systems", pp. 657–674. Springer (2008)
16. Kohavi, R., Provost, F.: Confusion matrix. Mach. Learn. **30**(2–3), 271–274 (1998)
17. Lavin, A., Ahmad, S.: Evaluating real-time anomaly detection algorithms–the Numenta anomaly benchmark. In: 2015 IEEE 14th International Conference on Machine Learning and Applications, ICMLA, pp. 38–44. IEEE (2015)
18. Mikalsen, M., Paspallis, N., Floch, J., Stav, E., Papadopoulos, G.A., Chimaris, A.: Distributed context management in a mobility and adaptation enabling middleware (MADAM). In: Proceedings of the 2006 ACM Symposium on Applied Computing, pp. 733–734. ACM (2006)
19. Mishra, A., Nadkarni, K., Patcha, A.: Intrusion detection in wireless ad hoc networks. IEEE Wirel. Commun. **11**(1), 48–60 (2004)
20. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference, pp. 305–319 (2014)
21. Phua, C., Lee, V., Smith, K., Gayler, R.: A comprehensive survey of data mining-based fraud detection research. arXiv preprint arXiv:1009.6119 (2010)
22. Roesch, M., et al.: Snort: lightweight intrusion detection for networks. In: LISA, vol. 99, pp. 229–238 (1999)
23. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. Trans. Auton. Adapt. Syst. (TAAS) **4**(2), 14 (2009)
24. Sama, M., Rosenblum, D.S., Wang, Z., Elbaum, S.: Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp. 261–271. ACM (2008)
25. Sterritt, R., Bustard, D.: Towards an autonomic computing environment. In: Proceedings of the 14th International Workshop on Database and Expert Systems Applications, pp. 694–698. IEEE (2003)
26. Strang, T., Linnhoff-Popien, C.: A context modeling survey. In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp, vol. 4, pp. 34–41 (2004)
27. Stubbs, J., Moreira, W., Dooley, R.: Distributed systems of microservices using docker and serfnode. In: 2015 7th International Workshop on Science Gateways, IWSG, pp. 34–39. IEEE (2015)
28. Wei, W., Fan, X., Song, H., Fan, X., Yang, J.: Imperfect information dynamic stackelberg game based resource allocation using hidden Markov for cloud computing. IEEE Trans. Serv. Comput. **11**(1), 78–89 (2016)

View publication stats