Learning - based Adaptation Framework for Elastic Software Systems

Yingcheng Sun
Case Western Reserve University
Cleveland, OH, USA
yxs489@case.edu

Xiaoshu Cai
Case Western Reserve University
Cleveland, OH, USA
xxc239@case.edu

Kenneth Loparo
Case Western Reserve University
Cleveland, OH, USA
kal4@case.edu

***Abstract*—Adaptation is a concern for elastic software systems. Conventional methods like Brownout try to deactivate optional computation per request after decoupling the software into different components, to lower the workload when peaks occur. However, resource-intensive components are not always easy to isolate, and some software systems are even not separable. In this paper, we propose a new paradigm that provides each core and mandatory component a corresponding alternative component, with similar function but lower resource consumption, and use reinforcement learning in a feedback loop control for the self-adaptation process. We modified the widely used benchmark RUBiS to make it weakly coupled and add alternative components. Experiments show that our framework dramatically improves both the efficiency and effectiveness of self-adaptation.**

***Index Terms*—adaptive software, reinforcement learning, programming paradigm**

I. Introduction

Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner. Self-adaptation is the most obvious characteristic of elastic software systems that enables systems to continuously adapt themselves to uncertainty in the environment. One of the challenges in such kind of systems concerns how to make adaptation to themselves at runtime dynamically in response to possible and even unexpected changes from the environment and/or user goals [1].

Different approaches have been proposed for the design of self-adaptive software, a prominent one being architecture-based adaptation [2]. For example, a paradigm called brownout [3] successfully controls the load balance by separating software components into mandatory and optional parts and adaptively activating or deactivating optional parts to manage resource usage in software systems. However, subordinate and resource-intensive components are not always easy to be isolated, and some software systems are even not separable, like single-function mobile apps with only few but highly correlated modules. We thus propose a new adaptation framework that separates software into three different types of components: mandatory, optional and alternative. The mandatory parts must be kept running all the time, such as the critical services in the system, including data-relevant services. The optional parts, on the other hand, need not be active all the time and can be deactivated temporarily to ensure system performance in the case of flash crowds. Compared to brownout programming paradigm [2] with only mandatory and optional components, we introduce a new type of component for the system design: alternative. Each mandatory component has its corresponding "alternative" one providing the same services but use less computation resources. Some "inseparable" software systems may not have "optional" parts, but definitely can build mandatory and corresponding alternative components.

DOI reference number: 10.18293/SEKE2019-009

After building self-adaptive software architecture, we also need to design a feedback loop control mechanism to fulfill the self-adaptation process. Control theory was used to adaptively determine when to activate/deactivate optional features in the applications through the feedback from software and environment, but applying control theory to adapt the software behavior is a complex problem [4], due to the difficulty of accurately modeling software, to the types of requirements and their tradeoffs [5] and to the need of instrumenting software to obtain sensor measurements and actuators [6]. Techniques from statistical machine learning have shown to be effective for feedback control in autonomic computing systems [7], and learning from system running environment can lead to improvements in accurately tuning parameters that avoids slow controller reactions to significant arrival rate changes [8] Therefore, in this paper, we propose a reinforcement learning based framework to cope with non-stationary environment and changeable user goals at runtime by learning controlling rules to find appropriate thresholds. We also designed the algorithm to compute the priority of component in an application, and modified the benchmark RUBiS to make it separable. Experiments show that our framework dramatically improves both the efficiency and effectiveness of self-adaptation.

II. Related Work

To enhance the adaptation of software system, the first step is to build the elastic software architecture with low-coupling characteristic, and then design a feedback loop control mechanism to fulfill the self-adaptation process. As discussed before, it is more efficient using machine learning than control theory as the control mechanism, so we only introduce existing works on feedback loop control through machine learning to support the online planning process of self-adaptive systems.

To make an application elastic, the designer needs to build a software architecture that can decompose the act of serving a request into different parts of the application, each dealing with a different part of the response. Some functions of a software application might be skipped when necessary. Rainbow [9] is a framework using reusable infrastructure to support runtime self-adaptation of software systems. Brownout [3] is a self-adaptive paradigm that enables or disables optional parts in the system to handle unpredictable workloads. In existing articles, the optional parts are identified as contents, components, and containers. Optional web contents on servers are to be showed selectively to users to save resource usage [10]. Components-based applications deactivate optional components to manage resource utilization [11]. In containerized clouds, each service is implemented as a container, and the optional containers can be activated/deactivated based on system status [12]. Optional parts might temporarily be deactivated so that the essential functions of the system are ensured and applications avoid saturation.

Several automatic policies based on machine learning and admission control were introduced. Desmeurs et al. [8] presented an event-driven brownout technique to investigate the tradeoffs between utilization and response time for web applications. Dupont et al. [13] proposed an automatic approach to manage cloud elasticity in both infrastructure and software. The proposed method takes advantage of the dynamic selection of different strategies. Moreno et al. [14] presented a proactive approach for latency aware scheduling under uncertainty to accelerate decision time, and applied a formal model to solve the nondeterministic choices of adaptation tactics. Li et al [15] [16] designed a multi-agents model to improve the self-adaptation of meta search systems.

Some existing works have the related idea with us to use Reinforcement learning (RL) to support the online planning process of self-adaptive systems. Amoui et al [17] use reinforcement learning to support action selection in the planning process and clarify why, how, and when reinforcement learning can be beneficial for an autonomic software system. Ho et al [18] present a model-based reinforcement learning approach that maintains a model to utilize the engineering knowledge and continuously optimizes system behavior through model-based reinforcement learning. Tianqi et al [1] combines reinforcement learning with case-based reasoning to overcome the limitations of rule-based adaptation in which decisions are only made based on static rules

The limitations of the past work are (i) the priority of software component is not discussed, and (ii) no strategy to deal with the inseparable components or applications. In next sections, we will discuss our proposed solutions.

III. LEARNING - BASED ADAPTATION FRAMEWORK

In this section, we first assign priority to each software component, and then propose a reinforcement learning based framework that supports self-adaptive activation/ deactivation of software components to avoid saturations.

A. Component Priority Assignment

Since some components of a system will be deactivated when unexpected peaks come, we want to know: which component should be deactivated first, i.e. how to determine the priority order of components that are deactivated? Two metrics are used to calculate the component priority: **computational complexity** and **usage frequency**. Computational complexity refers is the amount of resources required for running a component. The more complex a component, the more computing resources it needs. Usage frequency is the number of times that a component is invoked in the software system during a time interval. The algorithm of selecting the optional component is to choose the node with lower frequency but higher complexity first, because such kind of component needs more computing resources while it is less used by users. The more frequently a component being used, the more important and valuable it is. For component i, we have:

$$p = \delta.c/f$$

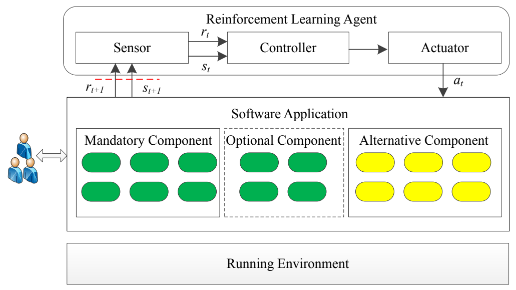
where p is the popularity of component i, δ is the control factor used for scaling component i's priority, c refers to the computational complexity and f is the usage frequency.

B. Reinforcement Learning based Adaptation Framework

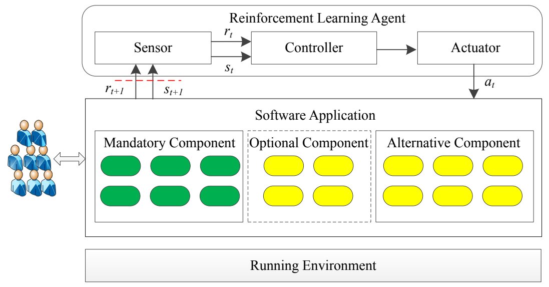
Conventional elastic software architectures usually separate software components into two parts: mandatory and optional, but not all software applications can be easily decoupled, so sometimes no components can be isolated. Another issue is what if a software still cannot adapt to the environment change after adjusting the running state of optional parts? For example, what if the workload of server is still high after deactivating the optional components? To deal with the above issues, we propose a new design paradigm that prepares alternative substitute for each of the mandatory components with similar functions but less complexity. Figure 1 illustrates the structure and working process of the proposed framework.

When the user traffic is low, the mandatory software components and the optional components are activated (if any), and the alternative parts are deactivated (Figure 1a). All functions are available and the complete service of the software is offered. When the traffic increases, the workload of server rises and the response time gets longer. In this case, optional components are deactivated in the order of priority. The user experience might be degraded, but the whole software will still work well instead of saturation. Figure 1b shows that all optimal components are deactivated.

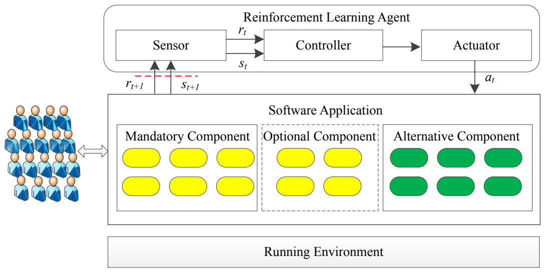
If the workload of server is still high after deactivating optimal components, we need to switch the service running on mandatory components to alternative components (Figure 1c). The alternative code provides similar but "lighter" services than mandatory content such as a website with static picture instead of dynamic animation. Sometimes applications may not contain components that can be grouped as "optional", like mobile apps with few functions, so alternative substitutes are necessary.



(a) Mandatory and optional components are activated



(b) Mandatory components are activated but optional components are deactivated



(c) Alternative components are activated

Fig. 1. The proposed reinforcement learning framework with three types of components: Mandatory, Optional and Alternative. The auto scaling process as the user traffic changes is depicted from subfigure (a) to (c)

To lower the maintenance effort, the whole feedback loop needs to be automatically managed. This would enable software applications to rapidly and robustly respond to environmental changes, and being self-adaptive. We use reinforcement learning to automatically acquire the optimal strategies as policies that controllers produce for different situations. In the reinforcement learning context, an agent takes action a_t when the system is in state s_t and leaves the system to evolve to the next state s_{t+1} and observes the reinforcement signal r_{t+1}. Decision making in elastic systems can be represented as an interaction between software controllers and environment through sensors and actuators, and the feedback reward is evaluated in the form of utility functions. An elastic system may stay in the same state, but should take different actions in different situations and workload intensity.

We study the adaptive management of resources task in which a Reinforcement Learning Agent (RLA) interacts with environment by sequentially choosing which software component to be activated or deactivated, so as to maximize its cumulative reward. We model this problem as a Markov Decision Process (MDP), which includes a sequence of states, actions and rewards. More formally, MDP consists of a tuple of five elements (*S, A, P, R,* γ) as follows:

- State space *S*: A state $s_t \in S$ is defined as the metric function that quantifies the degree of auto-scaling variables, such as workload, response time, throughout and etc. The items in s_t are sorted in chronological order. The elasticity policy is defined in terms of rules based on the metric function: ”IF workload is high AND response time is long THEN take action a_t”.
- Action space *A*: Each component has a running probability controlling how often it is executed, and the sum of running probability between a mandatory component and its corresponding alternative part should be 1. The action $a_t \in A$ of RLA is to change the probability of a component being activated. The action a_t is among three possible candidates:

$$a_t \in A = \{-\theta, 0, \theta\}$$

- Reward *R*: After the RLA taking action a_t at state s_t, i.e., changing the running probability of a component, the utility of software system changes and provides feedback, and the RLA receives immediate reward $r(s_t, a_t)$ according to the system's feedback. In this paper, we use the response time as the metric of system utility and have: .

$$r_t = \lambda(resp(t) - resp(t-1))$$

where *resp(t)* is the response time of the system at time *t*, and λ is a constant number scaling the reward value. If a controlling action leads to a decreased response time, the reward will increase, meaning the action is appropriate. Otherwise, if the reward is close to zero, it implies that the action is not appropriate.
- Transition probability *P*: Transition probability $p(s_{t+1}|s_t, a_t)$ defines the probability of state transition from s_t to s_{t+1} when RLA takes action a_t. We assume that the MDP satisfies:

$$p(s_{t+1}|s_t, a_t, ..., s_1, a_1) = p(s_{t+1}|s_t, a_t)$$

- Discount factor $\gamma : \gamma \in [0, 1]$ defines the discount factor when we measure the present value of future reward. In particular, when $\gamma = 0$, RLA only considers the immediate reward. In other words, when $\gamma = 1$, all future rewards can be counted fully into that of the current action.

The upper part of each graph in Figure 1 illustrates the agent-software interactions in MDP. With the notations and definitions above, the problem of adaptive management of resources can be formally defined as follows: Given the historical MDP, i.e., (S, A, P, R, γ), the goal is to find a controlling policy $\pi : S \to A$, which can maximize the cumulative reward for the software system. A widely-used reinforcement learning method is Q-learning that directly approximates the optimal quality function of a policy π:

$$Q^*(s,a) = maxQ^\pi(s,a)$$

and then derives the optimal policy from Q^* by selecting the highest valued action in each state:

$$\pi^* = argmax_{a \in A} Q^*(s,a)$$

To approximate the optimal Q-function, Q-learning repeats the following two steps: 1) choose action a at s using policy derived from the current Q-function; and 2) take action a, and then update Q-function with the observed reward using the following formula:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1},a) - Q(s_t,a_t)]$$

where a is the learning rate, and γ is the discount factor that determines the present value of future rewards.

IV. EXPERIMENT

A. Experiment Setting

We verify our framework based on a widely used benchmark e-commerce web application RUBiS, with the dataset "Oopsla paper dump" from RUBiS official website[1]. The evaluation setup includes a ThinkPad laptop with Intel Core i5 2.50 GHz processor and 10 GB RAM. This web application is deployed on the laptop, while the visiting traffic is generated by a load test tool named JMeter. JMeter allows to dynamically selecting the number of users and maintains a number of client threads equal to the number of users.

RUBiS benchmark implements three functions of an auction site: selling, browsing and bidding. There are two roles in this benchmark: server for an auction and client that can be a seller, buyer or visitor. To calculate the priority of each component, we assume that one file (*.class, .html or. xml*) represents one unit of complexity, which means if a component includes three files of *.class*, its complexity is three. Though it is not very accurate to compute the running time using this method, it has little effect to our project since the complexity of each file is close to the other. Therefore, we can view each file as a unit with the same resource requirement. We use "session mechanism" to identify users who are visiting the website and simulate the workload with the number of visits because it is not easy to get the actual workload of the server. When the user traffic changes, controllers can switch the running components dynamically.

We select "Recommendation for You" as the optional component according to the sorting result of priority value. In order to make a more obvious comparison, we add website effects to the page "AboutMe" to construct the mandatory component and its original plain webpage without any effects is regarded as the alternative part. The website effects are implemented by CSS and JavaScript. We also add a new alternative component "ViewItem_light" that introduces product items in words compared to the original indivisible "ViewItem" module that

[1]https://rubis.ow2.org/

TABLE I
SELECTED COMPONENTS FOR EXPERIMENTS

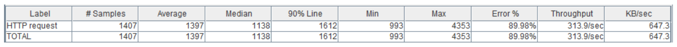
Component Type	Function	Description
Mandatory	AboutMe	AboutMe with web effects
	ViewItem	Introduction of products in pictures
Alternative	AboutMe_light	AboutMe without web effects
	ViewItem_light	Introduction of products in words
Optional	Recommendation	Recommendation for You

presents items in pictures. All the components used in our experiment are listed in Table 1.

For JMeter, three parameters need to be set before running the sampler: number of threads, ramp-up period and the number of times to execute the test. The ramp-up time tells JMeter how long to take to run full number of threads chosen. For example, if 10 threads are used, and the ramp-up period is 100 seconds, then JMeter will take 100 seconds to get all 10 threads up and running. Listeners added to the thread group are: "aggregate report", "view results tree" and "view results in table". "Aggregate report" shows results of measurements by calling the same page lots of times as if many users are calling that page. "Result tree" outputs the report in whether requests are responded successfully or not. In "result table", "sample time" means the time every request uses and "latency" indicates the time interval between request and response from the server.

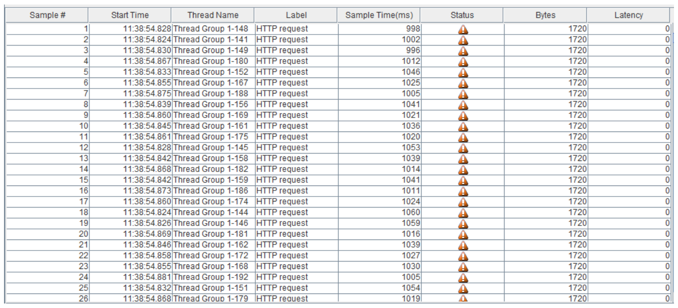
B. Evaluation

To verify the effectiveness of our proposed framework, two experiments are done in this section: running RUBiS in non-adaptive configuration and self-adaptive configuration. We first need to find the maximum capacity of our platform to acquire the configuration parameters for the "traffic peak". By gradually increasing the number of threads in the sampler, we obtain the lower limit of the peak condition: "launch 4000 threads in 4 seconds". Figure 2 shows the result.



Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP request	1407	1397	1138	1612	993	4353	89.98%	313.9/sec	647.3
TOTAL	1407	1397	1138	1612	993	4353	89.98%	313.9/sec	647.3

(a) Aggregate report with errors and high average time



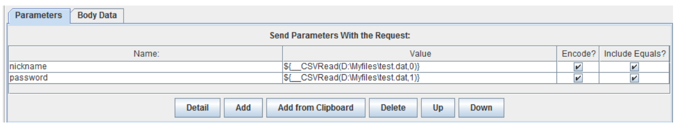
Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency
1	11:38:54.828	Thread Group 1-148	HTTP request	998	⚠	1720	0
2	11:38:54.824	Thread Group 1-141	HTTP request	1002	⚠	1720	0
3	11:38:54.830	Thread Group 1-149	HTTP request	996	⚠	1720	0
4	11:38:54.867	Thread Group 1-180	HTTP request	1012	⚠	1720	0
5	11:38:54.833	Thread Group 1-152	HTTP request	1046	⚠	1720	0
6	11:38:54.855	Thread Group 1-167	HTTP request	1025	⚠	1720	0
7	11:38:54.875	Thread Group 1-188	HTTP request	1005	⚠	1720	0
8	11:38:54.839	Thread Group 1-156	HTTP request	1041	⚠	1720	0
9	11:38:54.860	Thread Group 1-169	HTTP request	1021	⚠	1720	0
10	11:38:54.845	Thread Group 1-161	HTTP request	1036	⚠	1720	0
11	11:38:54.861	Thread Group 1-175	HTTP request	1020	⚠	1720	0
12	11:38:54.828	Thread Group 1-145	HTTP request	1053	⚠	1720	0
13	11:38:54.842	Thread Group 1-158	HTTP request	1039	⚠	1720	0
14	11:38:54.868	Thread Group 1-182	HTTP request	1014	⚠	1720	0
15	11:38:54.842	Thread Group 1-159	HTTP request	1041	⚠	1720	0
16	11:38:54.873	Thread Group 1-186	HTTP request	1011	⚠	1720	0
17	11:38:54.860	Thread Group 1-174	HTTP request	1024	⚠	1720	0
18	11:38:54.824	Thread Group 1-144	HTTP request	1060	⚠	1720	0
19	11:38:54.826	Thread Group 1-146	HTTP request	1059	⚠	1720	0
20	11:38:54.869	Thread Group 1-181	HTTP request	1016	⚠	1720	0
21	11:38:54.846	Thread Group 1-162	HTTP request	1039	⚠	1720	0
22	11:38:54.858	Thread Group 1-172	HTTP request	1027	⚠	1720	0
23	11:38:54.855	Thread Group 1-168	HTTP request	1030	⚠	1720	0
24	11:38:54.881	Thread Group 1-192	HTTP request	1005	⚠	1720	0
25	11:38:54.832	Thread Group 1-151	HTTP request	1054	⚠	1720	0
26	11:38:54.868	Thread Group 1-179	HTTP request	1019	⚠	1720	0

(b) View of results with failed request

Fig. 2. Peak evaluation

Figure 2a shows that the total number of successful samples is only 1407 because the web server cannot process all requests given only a limited time. The maximum response time set to 4s is because a study made by Amazon shows that 25% users will leave when the responding time is over 4s. The error rate is 89.98%. Errors are reflected by warning status in Figure 2b.

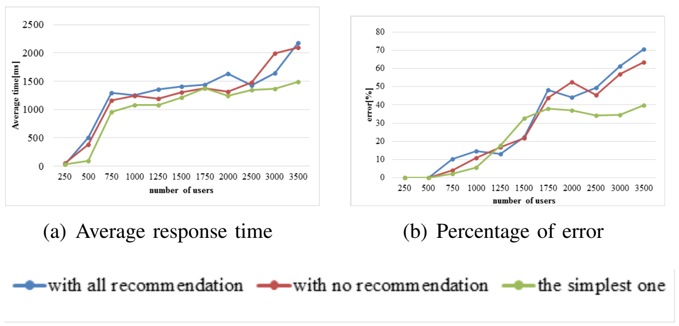
Next, we make comparison experiments between non-adaptive and self-adaptive patterns on the module ”About Me”. It askes to register first before accessing the page, so we package username and password information as a request and send it to the server. To make it close to the real scenario, we simulate different users to visit the web page instead of one user visit multiple times. The method is to put 100 users' information extracted from database into a “*.dat*” file with “nickname” and “password” as the parameters and package it as a request (See Figure 3).



Name:	Value	Encode?	Include Equals?
nickname	${__CSVRead(D:\Myfiles\test.dat,0)}	✓	✓
password	${__CSVRead(D:\Myfiles\test.dat,1)}	✓	✓

Fig. 3. Multi-users parameter. “Name” means the name list of variables used in the project. ”Nickname” is the first (No. 0) column in test.dat and ”password” is the second (No. 1) column in test.dat.

Non-adaptive Experiment: we observe the response time and error rate on three different running modes while the visits increase without adaptation mechanisms: 1) Mandatory + Optional components (“AboutMe” and “Recommendation”) 2) Mandatory component only (“AboutMe”) 3) Alternative component only (”AboutMe_light”). To evaluate the three modes, we set up samplers with the number of threads ranging from 250 to 3500 and launch in 4 seconds each time, and calculate the average response time and error rate for each sample. Figure 4 shows the performance of RUBiS with the three modes.

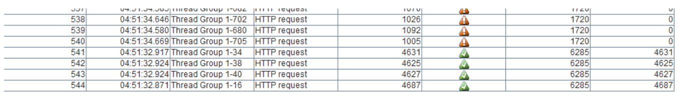


(a) Average response time
(b) Percentage of error

Fig. 4. Performance of RUBiS with three different modes.

Figure 4a shows that average response time increases with the number of visit users increases. It basically takes the longest time for a web server to support the functions in mode 1, because the “Recommendation” module involves frequent interactions with database. The performance improves a little after deactivating the “Recommendation” module in mode 2, and it improves obviously after switching the mode 2 to mode 3. In addition, Figure 4b shows that although some fluctuations appear in the graph, the trend is still very clear that the error rate tends to be flat when it is approaching to saturation after switching Mandatory component into an Alternative one.

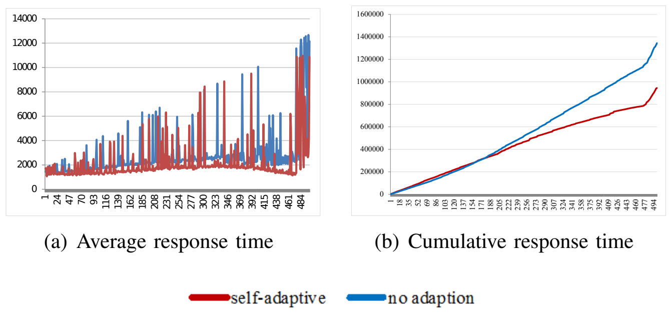
Self-Adaptive Experiment: to make the system self-adaptive, we first need to model the change of running environment. In this paper, we model the change of server workload. Since it is not easy to detect the real workload, we thus choose the response time as the metric, and the longer response time means the higher workload. However, we notice that the response time of failed requests are sometimes even shorter than successful ones in JMeter, which will interfere the whole process and make the result inaccurate. Figure 5 shows an example.



538	04:51:34.646	Thread Group 1-702	HTTP request	1026	⚠	1720	0
539	04:51:34.580	Thread Group 1-680	HTTP request	1092	⚠	1720	0
540	04:51:34.669	Thread Group 1-705	HTTP request	1005	⚠	1720	0
541	04:51:32.917	Thread Group 1-34	HTTP request	4631	✓	6285	4631
542	04:51:32.924	Thread Group 1-38	HTTP request	4625	✓	6285	4625
543	04:51:32.924	Thread Group 1-40	HTTP request	4627	✓	6285	4627
544	04:51:32.871	Thread Group 1-16	HTTP request	4687	✓	6285	4687

Fig. 5. Sample time of failed requests are shorter than successful ones

Due to the above property of JMeter, we evaluate the self-adaptive experiment by launching a limited number of threads to make sure there are no failed requests and the response time is in linear growth. We set the maximum number of threads to 500 and ramp-up period to 5 seconds, and we make sure there are no errors with the number of requests less than 500. We set the control factor δ to 1 for each component and set two elasticity policies for state space: “IF response time is over 3000 ms THEN take action deactivate Optional components.” and ”IF response time is over 6000 ms THEN take action *switch Mandatory components to Alternative parts*.” The whole self-adaptive process is controlled by the Reinforcement Learning Agent (RLA). Figure 6 shows the experiment results.

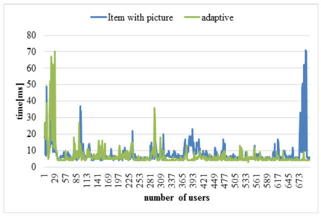


(a) Average response time
(b) Cumulative response time

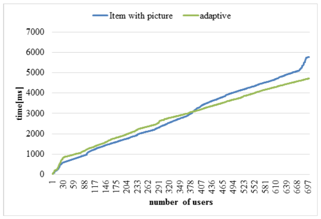
Fig. 6. Performance of running RUBiS with self-adaptive control in Mandatory, Optional and Alternative Components

When the number of visiting users is close to 150, the response time comes to 3000 ms that is our first threshold of ”high workload”. RLA captures the high workload through its sensor and take the action of deactivating the Optional components. However, we find that there is no obvious improvement in performance as shown in Figure 6. When the number of users is close to 200, the response time comes to 6000 ms that reaches our second threshold, so RLA takes the action of switching Mandatory components to their Alternative parts. We can see that the cumulative response time decreases dramatically compared to the system without self-adaptation mechanism from Figure 6b.

Next, we verify the effectiveness of our framework to deal with the problem of ”what if a software application cannot be easily decoupled?” When all the modules are highly correlated in a software application and cannot be easily isolated, the Brownout mechanism does not work anymore. As discussed before, we propose the method of switching the whole complex module or component into a simpler one, instead of isolating optional parts. As an example, we choose the function ”ViewItem” with all its components related and cannot be decoupled. We set the maximum number of threads to 700 and ramp-up period to 5 seconds. We set an elasticity policy for the state space: “IF response time is over 35 ms THEN take action *switch Mandatory components to Alternative parts*”. Figure 7 shows the experiment results.



(a) Average response time



(b) Cumulative response time

Fig. 7. Performance of running RUBiS with self-adaptive control in Mandatory and Alternative Components.

When the number of visiting users is close to 300, the response time comes to 38 MS that reaches our threshold, and then the action of switching the ”ViewItem” component to its Alternative part ”ViewItem_light” is taken. In Figure 7a, the response time is very high at the beginning because of the initialization of the module. After the modules switch, the response time of the system in self-adaptation mode is less than that without self-adaptation mechanism on average. From the cumulative response time shown in Figure 7b, we can see that it is very close for two curves at first, but the self-adaptive one outperforms the original one as the number of users increase.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a framework to support self-adaptation decision making. This framework separates software into three different types of components: mandatory, optional and alternative, and use reinforcement learning to design the ”controller” aiming at coping with non-stationary environment and changeable user goals at runtime. It will be interesting to integrate our framework with containers to improve scheduling performance in the future.

ACKNOWLEDGMENT

This work was supported by the Ohio Department of Higher Education, the Ohio Federal Research Network and the Wright State Applied Research Corporation under award WSARC-16-00530 (C4ISR: Human-Centered Big Data).

REFERENCES

[1] T. Zhao, W. Zhang, H. Zhao, and Z. Jin, “A reinforcement learning-based framework for the generation and evolution of adaptation rules,” in *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 103–112.

[2] D. Weyns, S. Malek, and J. Andersson, “Forms: Unifying reference model for formal specification of distributed self-adaptive systems,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 1, p. 8, 2012.

[3] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, “Brownout: Building more robust cloud applications,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 700–711.

[4] A. Filieri, H. Hoffmann, and M. Maggio, “Automated design of self-adaptive software with control-theoretical formal guarantees,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 299–310.

[5] K. Angelopoulos, A. V. Papadopoulos, and J. Mylopoulos, “Adaptive predictive control for software systems,” in *Proceedings of the 1st international workshop on control theory for software engineering*. ACM, 2015, pp. 17–21.

[6] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, “Control-theoretical software adaptation: a systematic literature review,” *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 784–810, 2018.

[7] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva, “Comparison of decision-making strategies for self-optimization in autonomic computing systems,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 4, p. 36, 2012.

[8] D. Desmeurs, C. Klein, A. V. Papadopoulos, and J. Tordsson, “Event-driven application brownout: Reconciling high utilization and low tail response times,” in *2015 International Conference on Cloud and Autonomic Computing*. IEEE, 2015, pp. 1–12.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-based self-adaptation with reusable infrastructure,” *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[10] J. Dürango, M. Dellkrantz, M. Maggio, C. Klein, A. V. Papadopoulos, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén, “Control-theoretical load-balancing for cloud applications with brownout,” in *53rd IEEE Conference on Decision and Control*. IEEE, 2014, pp. 5320–5327.

[11] F. Alvares, G. Delaval, E. Rutten, and L. Seinturier, “Language support for modular autonomic managers in reconfigurable software components,” in *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 271–278.

[12] M. Xu, A. V. Dastjerdi, and R. Buyya, “Energy efficient scheduling of cloud application components with brownout,” *IEEE Transactions on Sustainable Computing*, vol. 1, no. 2, pp. 40–53, 2016.

[13] S. Dupont, J. Lejeune, F. Alvares, and T. Ledoux, “Experimental analysis on autonomic strategies for cloud elasticity,” in *2015 International Conference on Cloud and Autonomic Computing*. IEEE, 2015, pp. 81–92.

[14] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, “Proactive self-adaptation under uncertainty: a probabilistic model checking approach,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 1–12.

[15] Q. Li and Y. Sun, “An agent based intelligent meta search engine,” in *International Conference on Web Information Systems and Mining*. Springer, 2012, pp. 572–579.

[16] Q. Li, Y. Zou, and Y. Sun, “Ontology based user personalization mechanism in meta search engine,” in *International Conference on Uncertainty Reasoning and Knowledge Engineering*. IEEE, 2012, pp. 230–234.

[17] M. Amoui, M. Salehie, S. Mirarab, and L. Tahvildari, “Adaptive action selection in autonomic software using reinforcement learning,” in *Fourth International Conference on Autonomic and Autonomous Systems (ICAS’08)*. IEEE, 2008, pp. 175–181.

[18] H. N. Ho and E. Lee, “Model-based reinforcement learning approach for planning in self-adaptive software system,” in *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*. ACM, 2015, p. 103.