Using Controllers to Adapt Messaging Systems: An Initial Experience

Nelson S. Rosa
nsr@cin.ufpe.br
Centro de Informática, UFPE
Recife, Pernambuco, Brazil

David J. M. Cavalcanti
djmc@cin.ufpe.br
Centro de Informática, UFPE
Recife, Pernambuco, Brazil

ABSTRACT

Adaptive middleware systems have been designed for various computing environments, including wireless sensor networks, IoT and cloud computing. Whatever the environment, however, the adaptation logic of these middleware systems rarely adopts control theory concepts. To shed some light on this topic, this paper presents the steps to using control theory in implementing an adaptive mechanism for a popular middleware model: message-oriented middleware, also known as messaging systems. These steps have been employed in a widely adopted open-source messaging system named RabbitMQ. This paper's contributions are on how to employ control theory step-by-step in messaging systems, along with some initial results on using P, PI and PID controllers.

KEYWORDS

Messaging system, Control Theory, Dynamic Configuration.

1 INTRODUCTION

The development of adaptive middleware is not a new topic in the middleware community [4]. Adaptive middleware systems are widely utilised in several application domains and computing environments, such as cloud [6], wireless sensor networks [14] and Internet of Things [5], just to mention the most important ones. Despite the diversity of existing adaptive middleware systems, developing control-theoretical solutions is not common in this research field.

However, similarly to other kinds of adaptive software systems [9, 15, 16, 18], control theory has become an candidate to be adopted to design adaptive middleware as an alternative to MAPE-K [12]. Most recently, "control" has been recognized as a new wave of research in self-adaptive systems [18] and can offer insights into the design of adaptation logic with formal guarantees.

In a simplified way, the use of control theory means that the software consists of a *feedback control loop* that includes a plant (e.g., the middleware), plant operation goals (e.g., *throughput* $>$ *50 messages/s*) and a controller that sets configurable plant parameters, e.g., size of the message queue used in the middleware. Adjustments (adaptations) are calculated by the controller using monitored data from the plant itself and the defined control targets. In practice, using control theory requires defining a mathematical model of the software and, more broadly, considering "controllability" as a first-class element in middleware design.

Adopting control theory can improve middleware development and execution with properties such as stability, quick convergence to its steady state, and accuracy. Those properties can work as design goals or be checked in the running middleware. On the other hand, the adoption of control theory has some associated challenges [16]: how to create a mathematical model of the middleware and its components, how to choose the controller to use, build middleware sensors and actuators, and how to ensure the goals of adaptation.

Considering the challenges, benefits, and the increasing use of control theory for software systems, this paper utilises this theory in a messaging system (also known as MOM (Message-Oriented Middleware)). This initial experience describes the steps to develop an adaptation logic grounded on control-theoretical concepts for the widely adopted messaging system RabbitMQ[1]. Finally, some initial results about the use of P (Proportional), PI (Proportional-Integral) and PID (Proportional-Integral-Derivative) controllers are also presented.

The remaining of this paper is organized as follows: Section 2 introduces basic concepts of RabbitMQ. Next, Section 3 presents the steps followed to instantiate the control theory to middleware. Section 4 presents an overview of existing solutions. Finally, Section 5 summarises the conclusion and future developments.

2 RABBITMQ

RabbitMQ is an open-source message broker widely adopted by enterprises and whose performance has often been evaluated in several ways [7][10]. A distributed application built atop RabbitMQ consists of publishers and consumers whose interactions are intermediated by a messaging broker. The broker stores messages on queues, manages subscribers, and route messages from publishers to consumers.

A vital aspect of these interactions is when the broker can remove messages from the queue, i.e., when the broker can consider the message handled by the consumer. Two approaches are available, and they depend on the acknowledgement mode used by the application. In the first case, a message is removed from a queue after the broker sends it (automatic ack). Another possibility is to remove the message from a queue after the consumer explicitly sends an acknowledgement to the broker (explicit ack).

The broker and consumers can be tuned at runtime in several ways. The broker tunning usually means configuring the Erlang virtual machine, e.g., garbage collection, process scheduler settings, and memory allocation. On the consumer side, it is possible to set a QoS parameter named *channel prefetch count* that defines the size of the *Prefetch Buffer*, as shown in Figure 1. This parameter specifies the max number of unacknowledged messages permitted on the channel. In practice, messages are cached in the consumer by the *RabbitMQ Client* until they are processed. It is worth observing that the broker delivers messages to consumers as fast as the network conditions or consumer will allow. Furthermore, by default, the prefetch buffer is unbounded.

[1]http://rabbitmq.com/

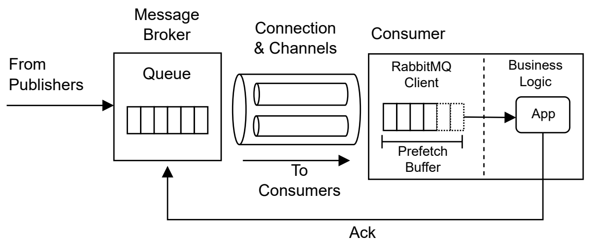

Figure 1: General overview of RabbitMQ elements

The best practice is to keep consumers busy while messages are stored in broker's queues and available for new consumers. The way to achieve this behaviour is by adequately configuring the prefetch count. This procedure may avoid thousand of messages stored in a given consumer while the queue is empty and new consumers have no access to delivered messages. Additionally, the configuration of the prefetch count may avoid overloaded consumers.

3 ADAPTATION UISNG CONTROL THEORY

The development of an adaptation solution using control theory can follow the general steps proposed in [8]. Initially, one needs to define the scope of what is to be modelled clearly. In this work, the *RabbitMQ Client* is the component of the consumer being controlled. It is worth noting that the consumer includes a client and the business logic, and the controller acts on the client to regulate the number of messages arriving in the business logic. A large number of messages can overload the business logic, while few messages lead to poor performance of the business.

Figure 2 shows a general overview of the closed loop associated with the RabbitMQ Client. After defining the scope, the next step is identifying a quantifiable system's goal to be tracked by the controller. Next, it is also needed to identify one or more knobs that change the system's behaviour and impact the goal. After defining the goal and knob(s), it is time to devise a mathematical model that captures their relationships. Next, the designer must specify the controller that acts on the knobs to close the system to the goal. Finally, the last steps consist of implementing and integrating the controller and testing and validating the system.

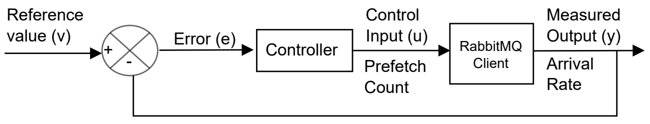

Figure 2: Closed loop

3.1 Identify goals and knobs

The design of an adaptation manager through a controller in the consumer helps to avoid flooding it with messages and keep the arrival rate close to the processing rate of the *App* (see Figure 1).

The metric associated with this goal is the *Arrival Rate*, which the consumer measures. It computes the number of messages arriving in the *App* per second after being stored in the *prefetch buffer*. It should be maintained close to the processing capacity of the business logic to keep the consumer in equilibrium.

The most direct way to control the *arrival rate* is to properly define the size of the *prefetch buffer* shown in Figure 1. This parameter is easily configured in RabbitMQ Clients and substantially impacts consumers' performance. A small value of *prefetch count* usually means fewer messages arriving in the *App* and larger queues in the broker. Meanwhile, a larger *prefetch count* may lead to consumer crashes due to memory run-out and empty queues in the broker.

3.2 Devise the model

After determining the scope, goal and knobs, the controller designer needs to devise a mathematical model able to quantify the effects of given control input (*prefetch count*) on a measured output (*arrival rate*). In practice, the model can be defined through linear difference equations representing the RabbitMQ Client's dynamics.

Due to the complexity of the consumer, the proposed model was defined as a black-box one that requires only knowledge about the relationship between inputs and outputs, as shown in Figure 2. This relationship can be fairly quantified by a simple linear differential equation that relates the *prefetch count* (u) that works as a control input and the consumer *arrival rate* (y) that is the measured output:

$$y(k+1) = ay(k) + bu(k) \tag{1}$$

This simple first-order model in the time domain (k is the time step) defines that the subsequent output, $y(k+1)$, depends only on the past input/output from a one-time unit.

According to [11], the following steps are necessary to refine this model: design experiments, estimate the model's parameters (a and b) and evaluate the model. If the model is acceptable according to some criteria (e.g., RMSE or R^2), proceed to design the controller. Otherwise, create new experiments until a proper model is found.

Design experiments. The construction of an accurate model needs enough data (training data) that allows a reasonable estimation of parameters a and b in Equation 1. The data should be generated through experiments that produce good-quality data. Parameters shown in Table 1 were used in the experiments.

Table 1: Parameters of the training

Range of control input (u)	[1,…,360]
Input signal behaviour	Step
Message size	256 (bytes)
No. of simultaneous publishers	25
Inter-publish time	Mean=10ms, stddv=1ms (Normal distribution)
Sample time	10 seconds
Business logic time	1 ms

The control input u was defined to explore several values of the prefetch count, which was increased step-wisely. The range of u was defined after some initial experiments that showed that values greater than 360 do not impact the arrival rate[2]. A short message size allowed to store millions of messages on the broker. The number of simultaneous publishers and inter-publish time have been set to generate a heavy workload on the broker. The sample time was specified to reduce noises commonly found in these kinds of experiments, e.g., the influence of the garbage collector. Finally, the business time was defined short so that the consumer quickly sends the ack to the broker. It is worth observing that different values should be used to explore other runtime scenarios or may be necessary for more complex environment setups, e.g., 500 simultaneous publishers or message sizes greater than 256 bytes.

The arrival rate is a linear function of the prefetch count until u becomes higher than 22, as shown in Figure 3. From this point, increasing the prefetch count does not linearly increase the arrival rate.

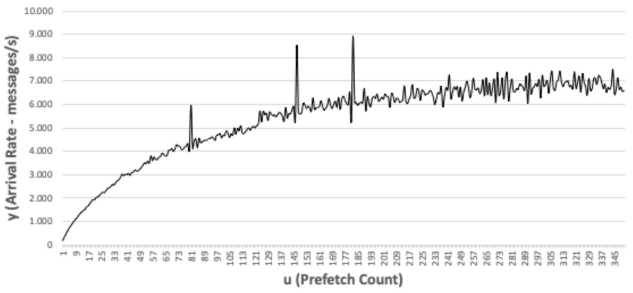

Figure 3: Observed data in open loop

Estimation of parameters. The least squares regression method was adopted using a set of training data [11] to estimate parameters a and b of Equation 1. The training data consists of 360 observations and are used to compute five quantities and the respective parameters a and b as defined in the following:

$$a = \frac{S_3S_4 - S_2S_5}{S_1S_3 - S_2^2}, b = \frac{S_1S_5 - S_2S_4}{S_1S_3 - S_2^2},$$

where S_1, S_2, S_3, S_4 and S_5 are computed as follows:

$$S_1 = \sum_{k=1}^{N} y^2(k), S_2 = \sum_{k=1}^{N} u(k)y(k), S_3 = \sum_{k=1}^{N} u^2(k),$$

$$S_4 = \sum_{k=1}^{N} y(k)y(k+1), S_5 = \sum_{k=1}^{N} u(k)y(k+1)$$

Model evaluation. Next, it is necessary to evaluate how good the model is, i.e., how the model explains the data observed. The accuracy of a model can be quantified based on the training data or a separate set of test data. A separate test data and two metrics were used in the evaluation, namely RMSE (Root Mean Square Error) and R^2 (Coefficient of determination) [11].

Using the training data, $a = 0.26$ and $b = 59.73$, Equation 1 becomes:

$$y(k+1) = 0.26y(k) + 59.73u(k). \tag{2}$$

[2]Data can be found at https://docs.google.com/spreadsheets/d/1HwCxo0XuX0t1QOZYeSCJ-TcmI39aV3-Z/edit?usp=sharing&ouid=102929903851857253702&rtpof=true&sd=true

According to this equation, the predicted *arrival rate* has a weak relation with the previous one (y) and a strong association with the *prefetch count* (u).

Table 2 summarises the evaluation of the devised model. It is worth observing that it was considered the linearity of the model in the range [1,22].

Table 2: Model evaluation

Prefetch count	(a,b)	Data	RMSE	R^2
[1,22]	(0.26, 59.73)	Training	8485.95	0.80
		Test	8586.84	0.82

Good models have $R^2 \geq 0.8$ [11]. As shown in this table, the Test Data have $R^2 = 0.82$, which means that the model defined in Equation 2 is a good approximation of the observed data.

Control Design. Then, it is time to define the controller. Three of the most popular controllers were designed: Proportional, Proportional-Integral, and Proportional-Integrative-Derivative.

Proportional Control (P): The proportional controller determines the value of $u(k)$ based on the following control law: $u(k) = K_p e(k)$, where $e(k) = r(k) - y(k)$ is the error, r is the goal and K_p (controller gain) is defined in the process of designing the proportional controller. This process starts by defining the transfer function of the closed loop having a proportional controller P. The transfer function is given by $F(z) = \frac{K_p b}{z-a+K_p b}$. By using the pole placement design [11], the stability condition of the this closed loop occurs in $\frac{a-1}{b} < K_p < \frac{1+a}{b}$, i.e., $-0.01234 < K_p < 0.02115$.

Proportional-Integral Control (PI): The control law of a PI controller is given by: $u(k) = u(k-1) + (K_p + K_i)e(k) - K_p e(k-1)$. It is worth observing that the computation of $u(k)$ requires knowing the values of current/past errors and control inputs. The transfer function of the closed loop with a PI controller is given by:

$$F(z) = \frac{b(K_p + K_i)z - bK_P}{z^2 + [b(K_p + K_i) - (1+a)]z + a - bK_p} \tag{3}$$

Similarly to the P controller, using the pole placement design, $K_p = \frac{a-0.36}{b}$ and $K_i = \frac{(a+K_p b)}{b}$, i.e., $K_p = -0.0016$ and $K_i = 0.0060$.

Proportional-Integrative-Derivative Control (PID): The control law of this controller is $u(k) = K_p e(k) + K_i \sum_{i=0}^{k-1} e(i) + K_i e(k) + K_d[e(k) - e(k-1)]$. The transfer function of the closed loop having a PID controller is defined as follows:

$$F(z) = \frac{b((K_p + K_i + K_d)z^2 - (K_p + 2K_d)z + K_d)}{D(z)},$$

where

$$D(z) = z^3 + (b(K_p + K_i + K_d) - (1+a))z^2 + \\ (a - b(K_p + 2K_d))z + bK_d$$

The solution of $D(z)$ for K_p, K_i and K_d yields $K_p = \frac{a-0.063-2bKd}{b}$, $K_i = \frac{0.3+a-bK_p-bK_d}{b}$ and $K_d = \frac{0.11}{b}$. For $a = 0.26$ and $b = 59.73$, the gains of this PID controller are $K_p = 0.0017761$, $K_i = 0.0058096$ and $K_d = 0.0018417$.

3.3 Implement and integrate the controller

As the controllers were designed, they were implemented in Go language and integrated into a RabbitMQ consumer. Additionally, it was necessary to instrument the consumer to collect the metric (arrival rate). This metric is collected from time to time (configurable) and is used to configure the next value of *prefetch count*. The next value of the prefetch count (control input) is calculated by the controller being used.

Next piece of code shows how the arrival rate is computed (Line 1), the value of the prefetch count is calculated (Line 2), and the new prefetch count is configured (Lines 3-7) inside the consumer:

```
   ...
1. s.ArrivalRate = numberOfMessages / monitorInterval
2. u = controller.Update(s.Ctler, s.ArrivalRate)
3. err := s.Ch.Qos (
4.            u, // new prefetch count
5.            0, // default
6.            false, // default
7. )
```

3.4 Test and validate the system

Figure 4 shows the setup environment in which the experiments were carried out. Publishers and consumers were implemented in Go Language and executed in different hosts, and the RabbiMQ broker was executed in a Docker container.

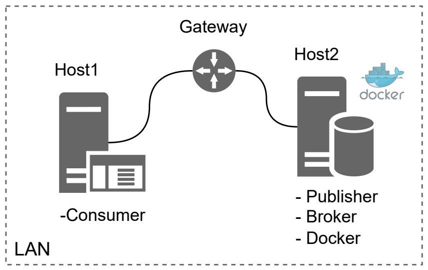

Figure 4: Experimental setup

Having implemented the controllers, the consumer's behaviour was observed considering them. Figures 5, 6 and 7 show the behaviour of the arrival rate having the goal (reference value) set to 400 msg/s (trend line), and whose closed loops are configured with P, PI and PID controllers, respectively. The mean arrival rates in these cases are 414.37 msg/s (P), 397.19 msg/s (PI) and 385.52 msg/s (PID), and their standard deviations are 216.09 msg/s (P), 95.12 msg/s (PI) and 196.03 msg/s (PD).

These figures show that the PI controller can maintain the arrival rate closer to the goal and should be selected to keep track of the arrival rate of a closed loop with a RabbitMQ consumer. A justificative for the best result of the PI controller comes from the fact that computing systems typically have a significant stochastic component, and the derivative control (D) may be more sensitive to this component.

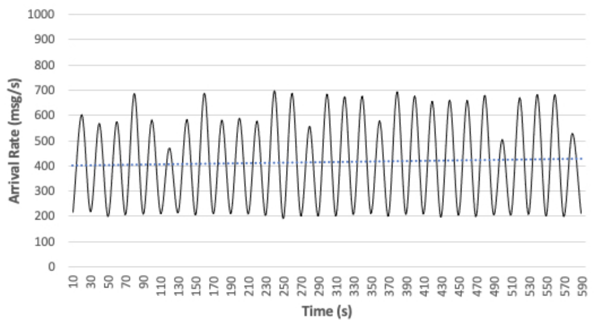

Figure 5: Closed loop using a P controller

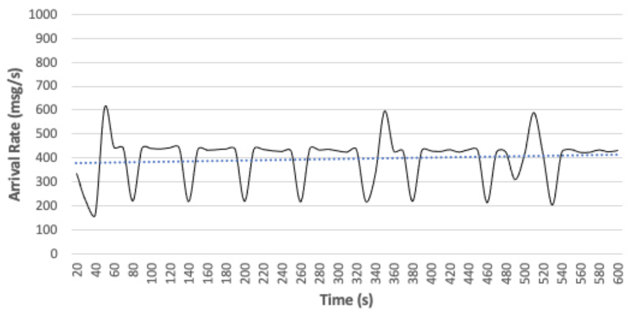

Figure 6: Closed loop using PI Controller

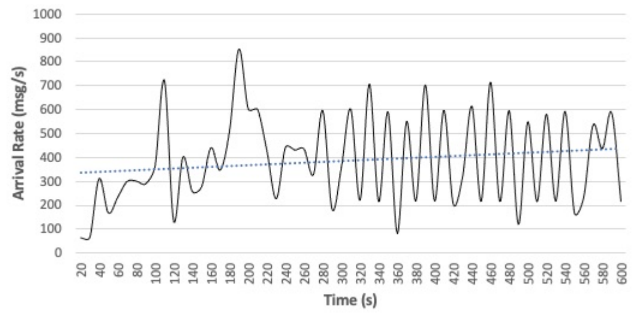

Figure 7: Closed loop using a PID controller

4 RELATED WORKS

Li and Nahrstedt [13] proposed a seminal work in this area. In their solution, the authors proposed a middleware capable of reconfiguring parameters and functionalities of a distributed multimedia application taking into account changes in CPU availability and execution environment bandwidth. The application behaviour is modelled through differential equations, and a PID controller is used. The controller keeps the number of requests arriving at the application close to a defined goal.

Having a focus on QoS (*Quality of Service*), ControlWare [19] is a middleware that uses control theory to provide performance guarantees for services available on the Internet. From the specification of performance goals by the users, the controllers, sensors and actuators necessary for the middleware are automatically generated so that it maintains the desired QoS. A P controller is used and customized according to defined performance goals. In turn, the sensors monitor CPU utilization metrics and queue size of service requests. Adaptations are performed with the controller changing the request queue management policy.

Abdelzaher [1, 2] also focuses on server performance control, specifically on Web servers. In this case, controllers are used to avoid overload and guaranteeing server performance when there is an unplanned increase in requests. The solution includes defining the behavioural model of server performance using a PI controller.

Like previous solutions, ACM [17] uses control theory to provide performance guarantees to distributed applications. The solution includes two PID controllers that regulate the *deadlines* for sending responses to client requests. Both controllers are associated with CPU utilization and bandwidth availability.

Unlike previous works, which present middleware systems for service-oriented applications, Banerje et al. [3] propose an adaptive middleware based on control theory for multimedia applications in wireless networks. In this case, the middleware uses an integral controller that defines the frequency of sending information (e.g., network traffic and resource availability) from the *streams* server to the application. The adaptations avoid network congestion and guarantee the level of QoS necessary for the application.

An initial difference between the related works and the focus of this paper is that they typically use control theory to reconfigure the resources the application uses. Nevertheless, they are not intended to reconfigure the behaviour of the middleware itself. Furthermore, they do not focus on messaging systems.

5 CONCLUSION, LESSONS LEARNED AND FUTURE WORKS

This paper presented the steps of applying control theory to a messaging system. These steps shows an initial experience on incorporate controllers to manage the execution of a messaging consumer. While grounded on mathematical principes, the whole solution was implemented in an existing commercial messaging system.

This initial experience have some learning points. Firstly, to properly design the experiments that are essential to define the devised model is a great challenge. Messaging systems are complex software systems whose set of configurable parameters is usually very big. Then, to identify the knobs (e.g., prefetch count) that have greater impact on what is being investigated (e.g., arrival rate) needs a good knowledge of the messaging system. Secondly, a first-order linear model is simple but facilitates enourmeously the use of control theory and helps to define the controllers. Finally, the integration of control elements into the messaging systems is facilitated by the proximity with the business logic.

Some important points should be taking in account in the future works. Firstly, the devised model only consider a given kind of workload. Then, it is necessary to devise new models considering workload profiles and then integrate them into the solution. Secondly, elements external to the closed loop are usually source of disturbances that should be also considered, e.g., a workload peaks. For example, the controller should be also capable of works on compensating these disturbances. Thirdly, the monitoring of the consumer may be subject to some kind of noise, e.g., delay in collecting the metric. Similarly to disturbances, the controller should be also designed considering measuring noises. Finally, it should be necessary to explore other kinds of controllers and formally check their properties.

REFERENCES

[1] T. Abdelzaher, Ying Lu, Ronghua Zhang, and D. Henriksson. 2004. Practical application of control theory to Web services. In *Proceedings of the 2004 American Control Conference*, Vol. 3. 1992–1997 vol.3.

[2] T.F. Abdelzaher, K.G. Shin, and N. Bhatti. 2002. Performance guarantees for Web server end-systems: a control-theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems* 13, 1 (2002), 80–96.

[3] N.L. Banerjee, K. Basu, and S.K. Das. 2005. Adaptive resource management for multimedia applications in wireless networks. In *Sixth IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks*. 250–257.

[4] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. 2001. The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online* 2 (June 2001), –.

[5] D. J. M. Cavalcanti and N. S. Rosa. 2021. Adaptive Middleware of Things. In *Proceedings of the 26th IEEE Symposium on Computers and Communications (ISCC 2021)* (Rennes, France) *(ISCC)*. IEEE Computer Society, New York, NY, USA, Article 1, 6 pages.

[6] Syed Muhammad Danish, Kaiwen Zhang, and Hans-Arno Jacobsen. 2021. BlockAIM: A Neural Network-Based Intelligent Middleware For Large-Scale IoT Data Placement Decisions. *IEEE Transactions on Mobile Computing* (2021), 1–1.

[7] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems* (Barcelona, Spain) *(DEBS 17)*. Association for Computing Machinery, New York, NY, USA, 227 – 238.

[8] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolas DIppolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. 2015. Software Engineering Meets Control Theory. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 71–82.

[9] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás Dippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro V. Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. 2017. Control Strategies for Self-Adaptive Software Systems. *ACM Trans. Auton. Adapt. Syst.* 11, 4, Article 24 (Feb. 2017), 31 pages.

[10] Guo Fu, Yanfeng Zhang, and Ge Yu. 2021. A Fair Comparison of Message Queuing Systems. *IEEE Access* 9 (2021), 421–432.

[11] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. 2004. *Feedback Control of Computing Systems*. John Wiley & Sons, Inc., Hoboken, NJ, USA.

[12] IBM. 2005. *An Architectural Blueprint for Autonomic Computing*. Technical Report. IBM.

[13] Baochun Li and K. Nahrstedt. 1999. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications* 17, 9 (Sept. 1999), 1632–1650.

[14] Jesús M. T. Portocarrero, Flávia C. Delicato, Paulo F. Pires, Taniro C. Rodrigues, and Thais V. Batista. 2016. SAMSON: Self-adaptive Middleware for Wireless Sensor Networks. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) *(SAC '16)*. 1315–1322.

[15] Bran Selic. 2020. Controlling the Controllers: What Software People Can Learn From Control Theory. *IEEE Software* 37, 6 (2020), 99–103.

[16] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. 2018. Control-Theoretical Software Adaptation: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 44, 8 (2018), 784–810.

[17] Xiao-An Shi, Xing-She Zhou, Xiao-Jun Wu, and Jian-Hua Gu. 2003. Adaptive control based dynamic Real-time resource management. In *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)*, Vol. 5. 3155–3159 Vol.5.

[18] Danny Weyns. 2017. Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges. In *Handbook of Software Engineering*. Springer.

[19] Ronghua Zhang, Chenyang Lu, T.F. Abdelzaher, and J.A. Stankovic. 2002. ControlWare: a middleware architecture for feedback control of software performance. In *Proceedings 22nd International Conference on Distributed Computing Systems*. 301–310.