Model-based System Identification for Cloud Services Analytics

Arun Adiththan
General Motors (Populus Group), Warren, MI 48092
Email: *arunadiththan@gmail.com*

Kaliappa Ravindran
City Univ. of New York, New York, NY 10031
Email: *ravi@cs.ccny.cuny.edu*

***Abstract*—** The issue of less-than-100% reliability and trustworthiness of third-party controlled cloud components (e.g., IaaS and SaaS components from different vendors) may lead to laxity in the QoS guarantees offered by a service-support system S to various applications. QoS laxity (i.e., SLA violations) may be inadvertent: say, due to the inability of system designers to model the impact of sub-system behaviors onto a deliverable QoS. Sometimes, QoS laxity may even be intentional: say, to reap revenue-oriented benefits by cheating on resource allocations and/or excessive statistical-sharing of system resources (e.g., VM cycles, number of servers). Our goal is to assess how well the internal mechanisms of S are geared to offer a required level of service to the applications. We use computational models of S to determine the optimal feasible resource schedules and verify how close is the actual system behavior to a model-computed 'gold-standard'. A cloud-based content distribution network (CDN) case study is described to illustrate our QoS assessment method.

I. Introduction

The QoS feature of a cloud-based system S depicts the ability of S to control its performance in response to an underlying infrastructure resource allocation or a change in the external environment conditions. The mapping between the output of S and platform resources should be known with reasonable accuracy: either as a closed-form model of S or through a series of incremental allocate-and-observe invocations on S [1].

The domain-specific core adaptation function in a cloud-based system S is viewed as a control-theoretic feedback loop acting on a reference input P_{ref}: say, for cloud resource management. Figure 1 concretizes this view. An external management entity H views the system S as supporting adaptation processes A'_p wrapped around a core system $g^*(I, O^*, s^*, E^*)$, i.e., $A'_p \otimes g^*(\cdots)$ — where '$\otimes$' denotes the composition in an object-oriented software view. Here, I, O^*, s^*, E^* denote the input, output, current state, and environment conditions, respectively. A'_p is embodied in a distributed agent-based software module that forms the building-block to structure S ([2] provides an architecture for distributed realization of the adaptation logic of A'_p). S interacts with its (hidden) external environment through the core elements $g^*(\cdots)$. Here, the meta-level signal flows between A'_p and $g^*(\cdots)$ are visible to H. The layered software structure of S intrinsic to cloud-based systems lends itself well for the dependability analysis by H.

The assessment module, H, has the following functions: (1) identifies plant state s such that the input action I initiated by

978-3-903176-15-7 © 2019 IFIP

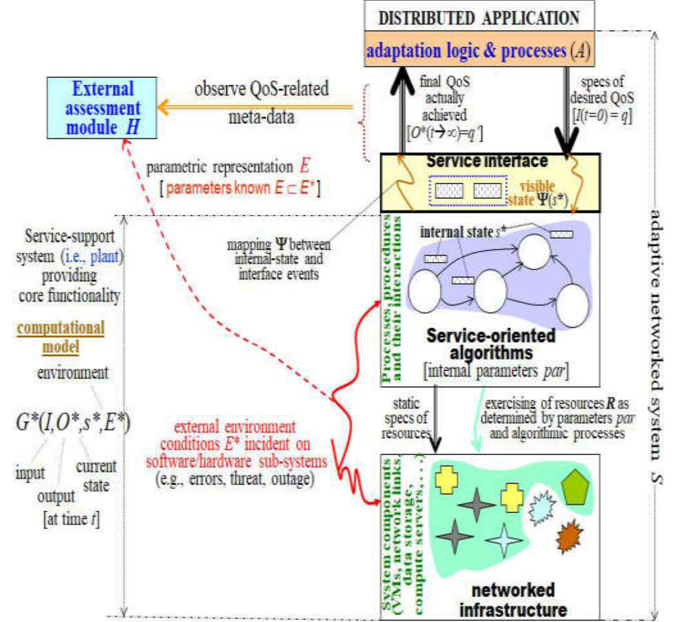

Fig. 1: Structure of adaptation process in a networked system

the controller in the subsequent iteration and the corresponding plant output O are accurately estimated; (2) reasons about the output tracking error $|P_{ref} - P'|$ under various environment conditions, and maps it onto a measure of the capability of S; (3) realizes adaptation of system S based on inferred system parameters, QoS deviation, and changes in the external environment conditions. This work primarily focuses on function #1 – system (or plant) identification – using techniques such as output behavior analysis of the actual system aided by the computational model under various parameter settings and system conditions to reason about the QoS compliance.

We delineate the QoS adaptation functionality embodied in a networked system from the monitoring of system-level meta-data and processes that capture the QoS behavior. In contrast, the existing assessment approaches (discussed in [3], [4]) are rigidly tied to the adaptation functionality of a target system (i.e., the adaptation logic deeply buried in the application layer) – and hence cannot be reused across other systems.

II. Our system identification approach

As described in Sec. I, a service-oriented view of a deployed cloud-based adaptive networked system encompasses

two sub-systems: (1) A core system, $g^*(\cdots)$, comprising of the infrastructure resources and components and a service-level algorithm that coordinates the infrastructure components; (2) An adaptation module $A[g(I, O^*, s, E), \phi]$ that embodies the control logic needed for adjusting the core system parameters.

The service providers' business practices do not allow them to expose the system-internal parameters s^*. Hence, we employ system identification techniques [5] to infer s^* based on externally observed output O^*. Basically, an inference about the unknown system parameters $(s^* - s)$ under a given knowledge about the environment condition $E \subset E^*$ aids in improving the accuracy of system model $g(I, O^*, s, E)$ — and hence the quality of QoS control.

A. System testing for internal parameter inference

Under the existing mode of service offering, the service providers may not expose their internal processes for reasons such as architectural limitations and/or lack of regulatory oversight. We envisage a futuristic business model for the evolving cloud-hosted services, wherein service providers' expose internal processes in the form of axiomatic specifications (without exposing proprietary service offering related information) to trusted entities. In the future, external assessment may even be mandated by regulatory agencies, as part of a service licensing and operating procedure (to exercise regulatory control over *service quality*). Our system identification technique makes use of the axiomatic specs and the training dataset for a representative parameter set and environment conditions exported to the external entity.

Fig. 2 shows the generic schema to infer system-internal processes and parameters using test inputs. During the service offering, the third-party entity collects and logs QoS meta-data.

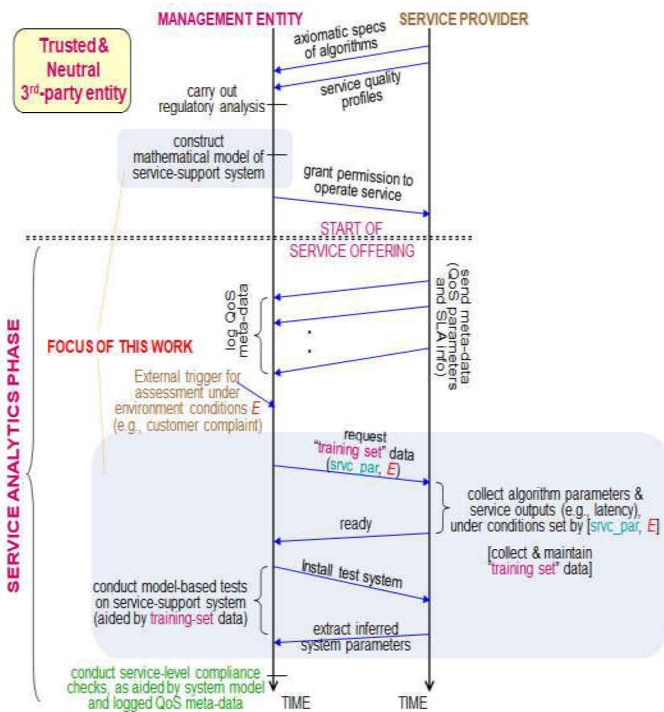

Fig. 2: System identification – generic schema

The management entity then uses the computational model, $g(I, O^*, s, E)$, of the service-support system for inferencing the system parameters. The trigger for assessment may arise due to a customer complaint regarding laxity in QoS or pre-scheduled assessment interval. The management entity obtains the training dataset from the provider and uses it to determine feasible test input ranges to test the system. The observed responses for the test input is then compared with the model-estimated output to infer the system internal parameters.

B. Feasibility evaluation of test inputs

For the purpose of making inference about the unknown parameters $(s^* - s)$, the tester module sends test sequences of inputs for which the system outputs are known from prior experiments under known environment conditions E — which constitute the "training dataset" in our approach. This testing with an actual system casts two important characteristics on the testing process itself: (1) The actual system $g^*(\ldots)$ is an integral part of the testing loop, which allows capturing the reactive capability of the system under test inputs; (2) To satisfy the stealthiness needs of testing, a pre-evaluation of the test inputs is carried out using the system model $g(\ldots)$ before exercising these inputs on the actual system $g^*(\ldots)$.

We refer to the pre-evaluation phase 2 in the testing process as "feasibility evaluation" of the test inputs. This phase ensures that the test inputs keep the actual system within normal operating regions (e.g., avoiding unusual resource congestions). This makes an actual system $g^*(\ldots)$ oblivious of differences between the tester-supplied inputs during test episodes and the application-level client supplied inputs during normal running phase. The inability of $g^*(\ldots)$ to distinguish between the test episodes and normal operations ensures the sanctity of service behavior offered by $g^*(\ldots)$.

The system-level testing methods proposed in this work enable identifying the unknown parameters $(s^* - s)$ — and hence the actual system's operating point, with high accuracy. In this light, the case study undertaken in this work is to infer the topological placement of content caching sites in a CDN. The design of controller logic $A[g(\ldots), \phi]$ and its impact on the quality of QoS adaptation is itself outside the scope of this work.

III. Case Study: Content Distribution Network

We consider a content distribution network (CDN) that delivers the contents maintained at a server R to various clients over a geographically spread-out cloud-based network topology. We describe our assessment method using a tree-based CDN as an example system structure. While a mesh-based CDN structure has advantages such as the ease of deployment and maintenance, a tree-based overlay structure for CDNs has its own advantages: such as the provability of QoS guarantees and the realization of bandwidth-efficient multicast for high-volume data content (e.g., video). From an assessment perspective, a declarative specs of system-internal algorithms (say, tree or mesh in a CDN) enables the structure-neutrality of our assessment methods.

Figure 3 shows layered-view of cloud-based CDN system. To meet the latency specs and fault-tolerance needs, the service-layer algorithm places proxy nodes in the distribution topology of S. The proxy nodes maintain a local copy of content p. A push/pull protocol extracts p from the proxy nodes for delivery to clients [6]. We assume a client-driven content pull algorithm (CL) in the SaaS layer. The CL algorithm determines if the content page at a proxy node is up-to-date using the local and global timestamp values associated with each page. If a local copy is out-of-date, the client request is propagated to upstream nodes.

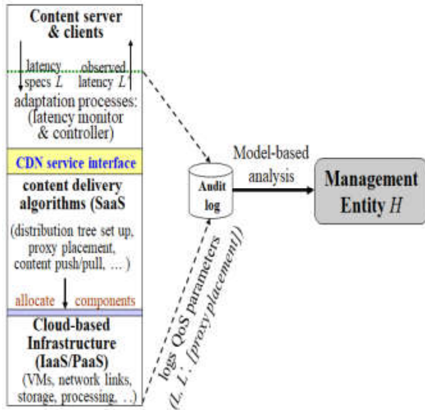

Fig. 3: Layered-view of cloud-based CDN realization

Our computational model of CDN is tractable due to its closed-form mapping of the proxy-placement parameters to per-pull latency, L, observed by the clients. The closed-form modeling of CDN arises from our consideration of a Poissonian arrival process for the client requests and the content updates at server R. This yields a set of queuing formulas for as building-blocks. In our earlier work [7], we presented a detailed discussion on queuing effects at proxy nodes and traffic flow analysis of the CDN system.

A. External assessment of CDN SaaS layer

The SaaS layer algorithmic externalization is essential for monitoring and enforcing QoS in a networked system. The externalization of a CDN system processes is basically a declarative specification of the various algorithms running in the CDN service layer. The system processes are basically the proxy placement algorithms on a distribution tree, generation of a suitable overlay tree on the CDN infrastructure, algorithms to update the contents at various proxy nodes, etc. The SaaS provider exposes a logical overlay tree for external assessment. The overlay tree comprises of the client locations and nodes that are part of the content distribution tree without revealing the actual placement of proxy nodes. The assessment module uses the information exposed by the service provider, training dataset, and the computational model of the CDN system to identify likely candidate placement scenarios and narrow down the search process using techniques such as trend analysis, client request rate perturbation, and Root Mean Squared Error (RMSE)-based statistical tool.

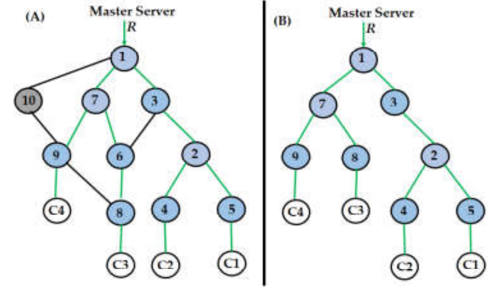

Fig. 4: Overlay tree topology: actual (A) vs. externalized (B)

L_{c1}	L_{c2}	L_{c3}	L_{c4}
0.11625	0.11644	0.08612	0.05779

TABLE I: Observed client latencies in the actual system

B. Illustration of proxy placement inference

We treat the discrete event simulation (DES) implementation (written in C) of the CDN system as the actual CDN system. The actual distribution tree realized on the cloud infrastructure and the CDN overlay tree externalized by the service provider for assessment is shown in Fig. 4. We consider identical storage and network link speeds for all nodes and links, for simplicity. The storage speed (μ_s) is set to 40 read/writes of content data per sec., and the network link speed (μ_n) is set to 12.5 content data transfers per sec. Content data is 2000 times larger relative to a control message used in CDN push/pull protocol. Data size is 400 Kbytes and control message size is 200 bytes. The storage speed and link bandwidth are 128 mbps and 40 mbps, respectively.

The inference process in a measurement epoch begins after collecting latency values from one run of the actual system. How the proxy nodes were setup in the actual system is not known to the assessment module. The assessment module upon receiving the latencies invokes the proxy placement inference algorithm. In the sample inference process described below, we use the client latencies obtained for the CDN distribution tree shown in Fig. 4 (B). Table I shows the observed latency values for all 4 clients for a claimed number of 3 proxy node configuration by the service provider.

As part of the evaluation process, the assessment module tries out different number and placement of proxy nodes. Only after a candidate placement of proxies is verified as feasible (say, based on transactional load and utilization of the proxy node), the external entity proceeds to evaluate the performance of the system. The values for server update rate (λ_s) and client request rate (λ_c) are available via the service interface of the SaaS-layer algorithm.

Table II the client latencies for 17 different proxy placements explored by the inference algorithm with [$\lambda_s = 0.1, \lambda_{c1,c2,c3,c4} = 2.0$]. After obtaining the latency values for the potential proxy placements, the inference process explores the right & left branches of the tree in steps 1 & 2, respectively. If the difference in the model-computed and observed latencies deviate within the threshold, $devThresh$, then a particular

Config. Id	Proxy Nodes	L_{c1}	L_{c2}	L_{c3}	L_{c4}
P.4.1	{1,4,5,7}	0.03008	0.03008	0.08471	0.05653
P.4.2	{1,3,8,9}	0.08471	0.08471	0.03008	0.02881
P.3.1	{1,2,7}	0.05717	0.05717	0.08471	0.05653
P.3.2	{1,3,7}	0.08471	0.08471	0.08471	0.05653
P.3.3	{1,2,9}	0.0572	0.0572	0.11106	0.02887
P.3.4	{1,5,9}	0.03021	0.11255	0.11255	0.02894
P.3.5	{1,2,8}	0.0572	0.0572	0.03014	0.08288
P.3.6	{1,5,7}	0.03014	0.11106	0.08474	0.05656
P.3.7	{1,4,5}	0.03021	0.03021	0.11255	0.08436
P.3.8	{1,4,9}	0.11255	0.03021	0.11255	0.02894
P.3.9	{1,8,9}	0.11255	0.11255	0.03021	0.02894
P.2.1	{1,8}	0.11412	0.11412	0.03029	0.08594
P.2.2	{1,7}	0.11247	0.11247	0.08478	0.0566
P.2.3	{1,3}	0.08478	0.08478	0.11247	0.08429
P.2.4	{1,9}	0.11412	0.11412	0.11412	0.02901
P.2.5	{1,2}	0.05723	0.05723	0.11247	0.08429
P.1	{1}	0.11589	0.11589	0.11589	0.08771

TABLE II: Model-based trend analysis for inference

L_{c1}	L_{c2}	L_{c3}	L_{c4}
0.11692	0.11671	0.09691	0.0657

TABLE III: Observed client latencies after perturbation

placement is deemed as a likely actual placement scenario. The $devThresh$ value 0.03 is a learned parameter using RMSE tools on the training data set externalized by the service provider. The RMSE values between the model-estimated and observed latencies for the initially considered 17 placements are as follows: 0.06101, 0.03867, 0.04185, 0.02239, 0.04597, 0.0473, 0.05186, 0.04315, 0.06372, 0.04738, 0.03157, 0.0313, 0.00288, 0.02911, 0.02014, 0.04579, 0.02111. At the end of step 2, based on the deviation threshold, 5 out of 17 placements are identified as likely, as highlighted in gray color in Table II.

Config. Id	Proxy Nodes	L_{c1}	L_{c2}	L_{c3}	L_{c4}
P.3.2	{1,3,7}	0.08471	0.08471	0.09508	0.06429
P.2.2	{1,7}	0.11247	0.11247	0.0951	0.06432
P.2.3	{1,3}	0.08491	0.08491	0.12589	0.09511
P.2.4	{1,9}	0.11701	0.11701	0.12481	0.02969
P.1	{1}	0.12315	0.12315	0.13095	0.10017

TABLE IV: Perturbation analysis using the system model

In step 3, we employ perturbation analysis technique to further narrow down the likely placement search. Here, the request rates of clients on one branch of the tree are perturbed by δ and observe the impact on latencies for all four clients. In our example, we perturb the request rates of C3 and C4 by 3.0, i.e., $\delta = 3.0$ and the latencies from both the actual system and analytical model are obtained and noted in Tables III and IV. In the actual system, the percentage increase in latency (with and without rate perturbation) for C1, C2, C3 and C4 are 0.576, 0.232, 12.529 and 13.688 respectively. The close-to-similar increase indicates that C3 and C4 share an intermediate proxy node in the actual system. Also, the minimal impact on C1, C2 indicates that all four clients do not have a common

Config. Id	Proxy Nodes	L_{c1}	L_{c2}	L_{c3}	L_{c4}
P.3.2	{1,3,7}	0.08471	0.08471	0.09508	0.06429
P.2.2	{1,7}	0.11247	0.11247	0.0951	0.06432
P.2.4	{1,9}	0.11701	0.11701	0.12481	0.02969
P.1	{1}	0.12315	0.12315	0.13095	0.10017

TABLE V: Model-based statistical analysis

intermediate proxy node. Now, among the model-computed latencies we eliminate the scenarios where we don't observe a similar latency change trend as that of the actual system. The likely placements at the end of step 3 in the inference process is highlighted in gray in Table IV. The corresponding RMSE values between the model-estimated and observed latencies are: $0.02273, 0.00328, 0.02278, 0.02463$ respectively.

The RMSE value is again computed between the observed and model-computed latency values obtained with a new perturbation. Table V shows the likely placements at the end of step 3. In this step, we choose the candidate proxy placement that incurred the lowest RMSE value for the model-computed and actual client latencies under rate perturbation. Thus, we have configuration P.2.2 with proxies at node 1 and 7 is the only candidate remaining at the end of step 3. Alternatively, a test designer may choose to select the lowest two or three configurations as likely. Such a threshold setting may, however, have an impact on the inference accuracy.

IV. Conclusion

Our generalized QoS assessment process brings advantages in cloud-based system software development. First, system management functions to realize QoS assessment can be employed across different application domains. Second, the domain-specific QoS characterization & enforcement mechanisms can evolve independently from the QoS assessment procedures that evaluate the domain-specific mechanisms. This is in contrast with the existing software engineering methods that rigidly integrate QoS management into the domain-specific system functionality. As a case study, we discussed the model-based QoS assessment of a cloud-hosted CDN service.

References

[1] C. Lu, Y. Lu, T. F. Abdelzaher, J. A Stankovic, and S. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, 2006.

[2] P. G. Bridges and R. D. Hiltunen, M. A .and Schlichting. Cholla: A framework for composing and coordinating adaptations in networked systems. *IEEE Transactions on Computers*, 58(11):1456–1469, 2009.

[3] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang. Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, 5(1):11, 2014.

[4] M. H. Ghahramani, M. Zhou, and C. Hon. Toward cloud computing qos architecture: Analysis of cloud systems and cloud services. *IEEE/CAA Journal of Automatica Sinica*, 4(1):6–18, 2017.

[5] K. J. Åström and P. Eykhoff. System identification - a survey. *Automatica*, 7(2):123–162, 1971.

[6] J. Kangasharju, J. Roberts, and K. W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4):376–383, 2002.

[7] K. Ravindran, K. Fayzullaev, and Y. Wardei. Model-based techniques for qos assessment of cloud-hosted cdn services. In *IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–6. IEEE, 2016.