2013 IEEE Sixth International Conference on Cloud Computing

DR^2: Dynamic Request Routing for Tolerating Latency Variability in Online Cloud Applications

Jieming Zhu[†], Zibin Zheng[†‡], and Michael R. Lyu[†]

[†]Shenzhen Research Institute, and Department of Computer Science & Engineering,
The Chinese University of Hong Kong, China
[‡]State Key Lab for Novel Software Technology, Nanjing University, China
{jmzhu, zbzheng, lyu}@cse.cuhk.edu.hk

***Abstract*—Application latency is one significant user metric for evaluating the performance of online cloud applications. However, as applications are migrated to the cloud and deployed across a wide-area network, the application latency usually presents high variability over time. Among lots of subtleties that influence the latency, one important factor is relying on the Internet for application connectivity, which introduces a high degree of variability and uncertainty on user-perceived application latency. As a result, a key challenge faced by application designers is how to build consistently low-latency cloud applications with the large number of geo-distributed and latency-varying cloud components. In this paper, we propose a dynamic request routing framework, DR^2, by taking full advantage of redundant components in the clouds to tolerate latency variability. In practice, many functionally-equivalent components have been already deployed redundantly for load balancing and fault tolerance, thus resulting in low additional overhead for DR^2. To evaluate the performance of our approach, we conduct a set of experiments based on two large-scale real-world datasets and a synthetic dataset. The results show the effectiveness and efficiency of our approach.**

***Keywords*—Cloud computing; latency variability tolerating; request routing; latency prediction; component selection**

I. Introduction

Cloud computing, as an Internet-based virtual computing environment [1], has recently gained much popularity for provisioning shared configurable resources (*e.g.*, infrastructure, platform, and software) as services on demand over the Internet [2]. However, different from hosting applications across a local enterprise network, when deploying or migrating applications to the cloud, many of them may break down due to networking delays of tens or hundreds of milliseconds [3]. The application latency in the cloud usually presents high variability over time. It has become an urgent yet challenging task to build fluidly responsive cloud applications, especially for latency-sensitive applications.

Application latency (*i.e.* response time) stands for the time duration from a user sending out a application request to receiving a response, which is one significant user metric for evaluating the performance of online cloud applications. Application latency has a tremendous effect on the user experience. For example, according to the report in [3], a half-second delay will cause a 20% drop in Google's traffic, and a tenth of a second delay can cause a drop in one percent of Amazon's sales.

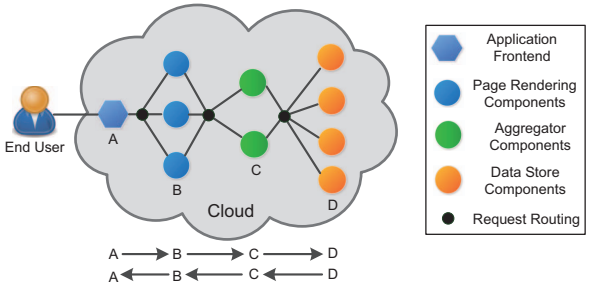


Fig. 1. A Page Request Example in Amazon

Nowadays there are many cloud applications hosted online for user interactions, such as search engine, e-commerce, social network, etc. These online cloud applications are large-scale and complex in system structures, which typically involve a lot of cloud components interacting with each other by using communication mechanisms like remote procedure call (RPC) or message-centric protocols [4]. For user interactions, each application request encompasses a large number of interactions between cloud components, in some cases across hundreds of machines. Fig. 1 depicts an example, taken from [5], of a page request to one of the e-commerce sites in Amazon. To generate the dynamic Web content, each request typically requires the page rendering components to invoke other aggregator components, which in turn query some other data store components to construct a composite response. These components are dependent between each other, and thus it is common to have a large call graph of an application [5]. To ensure that the page rendering engine can provide fluid response for maintaining seamless page delivery, the call chain between components must have consistently low latency.

However, latency between cloud components usually experiences high variability with a long tail [6]. There are a lot of subtleties that influence the latency in the cloud, including virtualization, shared network links, etc. Among them, one of the most important impact factor is relying on the Internet for application connectivity, which introduces a high degree of variability and uncertainty on user-perceived application latency. For example, as the size and complexity of the system scale up, the components in an Internet-scale cloud application have to be deployed across multiple geographically distributed data centers (*e.g.*, [7], [8], [9]), in order to locate

978-0-7695-5028-2/13 $26.00 © 2013 IEEE
DOI 10.1109/CLOUD.2013.59

resources closer to end-users. As a consequence, there is an increasingly urgent need for latency variability tolerating techniques to build consistently low-latency cloud applications with the large number of geo-distributed and latency-varying cloud components.

There are usually a lot of redundant components in the cloud. For example, for the purpose of load balancing and reliability guarantee, each component is typically deployed as multiple instances, which may be geographically distributed in the cloud. As such, the motivation of this paper is to take full advantage of these redundant and geo-diverse components to create a consistently low-latency application out of less-consistently responsive components for serving users world-wide, just as fault-tolerant techniques aim to create a reliable whole out of less-reliable parts [6].

To address this problem, in this paper, we propose a dynamic request routing framework, DR^2, to tolerate latency variability in cloud applications. The basic idea of DR^2 is to make dynamic selection among available redundant component when routing a traffic of application requests from different end-users. In this way, although the latencies between cloud components present high variability, we aim at minimizing the application latency of the whole call graph for each request, thus tolerating the low-level component-component latency variability. Towards this end, our dynamic request routing framework jointly performing online latency prediction between components and adaptive component selection among redundant candidates. In the online latency prediction phase, an online matrix factorization model is proposed to incrementally adapt our model to newly observed latency data, and then make accurate predictions for other unobservable values. In the adaptive component selection phase, we periodically re-optimize the component selection strategy to take the newly updated prediction values into account. More specifically, our main contributions include:

1) We characterize the latency variability problem in cloud computing systems and propose a dynamic request routing framework, DR^2, to tolerate this variability for building consistently low-latency cloud applications. Our approach takes full advantage of available redundant components, resulting in low additional overhead.

2) Our DR^2 framework is performed as a two-phase procedure: online latency prediction and adaptive component selection, in which online matrix factorization model and directed acyclic graph (DAG) based linear-time shortest path algorithm are integrated to achieve this goal.

3) Extensive experiments are conducted based on two real-world datasets and a large synthetic dataset to evaluate the effectiveness and efficiency of our DR^2 framework. The experimental results show that DR^2 not only can effectively tolerate the latency variability in online cloud applications, but also has fast convergence in online latency prediction and high scalability in adaptive component selection, which substantially outperforms other approaches.

Paper Organization. Section II introduces the motivation of this paper. Section III presents the system architecture. Section IV describes our DR^2 approach in detail. The experimental results are reported in Section V. We discuss the related work in Section VI and finally conclude this paper in Section VII.

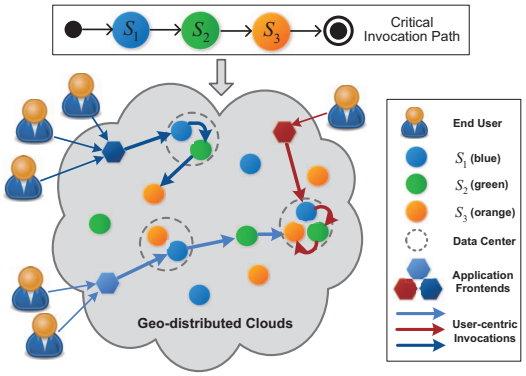


Fig. 2. A Prototype of Dynamic Request Routing

II. Motivation

A. A Prototype of Dynamic Request Routing

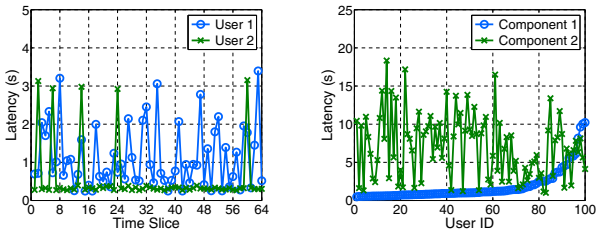
To clarify our motivation in this paper, we illustrate a prototype of dynamic request routing in Fig. 2. As shown in the figure, a critical invocation path is given to define the invocation dependencies between a set of tasks (S_1, S_2, S_3), where each task can be completed by a corresponding cloud component. A critical path is typically the invocation path that determines the whole application latency, and can be identified from the application logic manually by engineers or using some automatic detection techniques. However, how to construct the critical path is out of the scope of this paper.

For each component, there are many redundant instances deployed in the cloud, which may be across multiple data centers. To improve the user experience when serving globally-distributed users, both application frontend servers and cloud components are placed at multiple geographically distributed locations to make the resources closer to end users. End users from different regions are usually directed to the closest application frontend server when sending an application request. Given a stream of application requests from different end users, our objective is to minimize the user-perceived application latency and tolerate the latency variability by performing dynamic request routing for each specific request from each frontend server. For example, the invocation paths, depicted in arrows with different colors, illustrate different request routing strategies while minimizing the length of each path.

B. Main Challenges

Although similar works exist for request routing in content distribution networks [10] or Web service selection [11], these approaches are not sufficient to address our problem. The context of component selection and request routing in cloud computing have posted some new challenges, which are described as follows:

1) ***Latency Variability****:* As mentioned before, relying on the Internet for connectivity between cloud components introduces high variability on latency. Fig. 3(a) depicts a concrete example, where user 1 and user 2 are randomly selected from a real-world component latency dataset (dataset2 in our experiments). Both users have high latency variability to the same cloud component over the 64 time slices, where the interval



(a) Latency v.s. Time Slice (b) Latency v.s. User ID

Fig. 3. Latency Variability (Data from dataset2 in Section V-A2)

between time slices is 15 minutes. As a result, a key challenge faced by application designers is how to build consistently low-latency cloud applications by employing the large number of distributed and latency-varying cloud components.

*2) **Adaptivity**:* In large-scale cloud systems, the compute environment always change over time, due to 1) the shared nature of resources and network links, and 2) the diverse geo-distribution of cloud components. In face of the highly dynamic environment, it is vital for the request routing approach to be performed in an online fashion and be adaptive to the changes over time.

*3) **User Centricity**:* As illustrated in Fig. 2, to better serve users world-wide, the application frontend servers are also deployed as multiple instances dispersed at different regions. Users at different locations will experience quite different latency performance even to the same component considering the network latency. Fig. 3(b) shows an example, where we randomly select 100 users and 2 components from the dataset2, and evaluate the latencies from users to the same component. For clarification, we sort the user ID by the latencies to component 1. Large variations between users are observed in this figure. Hence, to provide fluid responsiveness to the users, the component selection strategy needs to be centric to each user.

*4) **Scalability**:* As the size and complexity of cloud applications scale up, more and more components and users will be added. Thus, it is crucial to be scalable on the length of critical path and the large number of cloud components. In addition, due to the high latency variability, the component selection algorithm needs also be efficient enough to be performed periodically.

III. System Architecture

In order to build seamlessly responsive cloud applications, we propose a dynamic request routing framework by taking advantage of the redundant components to tolerate the latency variability. Our framework is adaptive, user-centric and scalable, which addresses all the challenges mentioned above.

The framework is illustrated in Fig. 4, which includes the following two phases:

*1) **Online Latency Prediction**:* In order to make dynamic request routing, the precondition is to obtain real-time latency data between components. To overcome the overhead of active measurement, we resort to online latency prediction. First, request logs are collected to passively obtain the historical

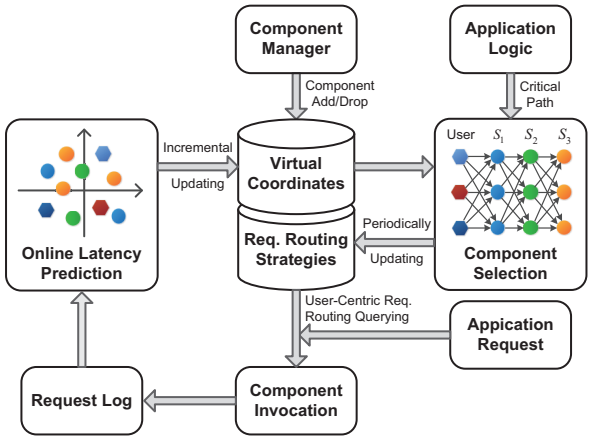


Fig. 4. The Framework of Dynamic Request Routing (DR^2)

latency data. Then a matrix factorization model is trained using newly updated historical data in an online machine. That is, we assign each user or component a virtual coordinate, and incrementally update the corresponding coordinate through an online learning algorithm by using each data sample. In this way, our approach can adapt to the changes over time. The updated virtual coordinates can then be used to predict the latency.

*2) **Adaptive Component Selection**:* Given a critical invocation path, performing dynamic request routing is to make adaptive component selection for each task. First, we build an invocation graph of available candidate components using the predicted pairwise latencies. Then, we propose an efficient shortest path to find the optimal component selection strategy for each user, which takes advantage of the graph characteristics (*e.g.*, DAG). Our algorithm works on a virtual graph which is transformed from the original invocation graph, thus reducing much overhead for multiple-user scenarios. The component selection needs to be performed periodically to adapt to the changing latency, and the results are stored in the database as request routing strategies.

When receiving an application request, the end user will be directed to one of the frontend server, and then a request routing strategy can be obtained by querying the database. After completing the request, the historical data of component invocations can be collected to update the matrix factorization model again.

IV. DR^2 Approach

Our approach of dynamic request routing is performed as a two-phase procedure: online latency prediction and adaptive component selection.

A. Phase 1: Online Latency Prediction

The basic idea of latency prediction is to use the historical data to predict the unknown values. Normally, suppose there are n users and m services in our cloud application, we can obtain two data matrix $L^U \in \mathbb{R}^{n\times m}$ (latencies between users and components) and $L^S \in \mathbb{R}^{m\times m}$ (latencies between components). Note that both matrices are asymmetric, since the

	S_1	S_2	S_3	S_4
U_1	0.2	?	0.3	?
U_2	?	0.3	?	0.4
U_3	0.4	0.6	?	?
U_4	?	?	0.8	0.6

(a) L^U Matrix

	S_1	S_2	S_3	S_4
S_1	0	?	0.7	?
S_2	?	0	?	0.8
S_3	0.5	0.6	0	?
S_4	?	0.4	0.3	0

(b) L^S Matrix

Fig. 5. An Example of Latency Prediction

latency $a \to b$ and $b \to a$ can be different due to the routing policy in the Internet. Fig. 5 illustrates an example, where values in grey are historical data, and the blank values are to be predicted. However, in each request a user usually invokes only a small set of components. Besides, only newly updated data can be used to get the real-time latency predictions, thus L^U and L^S are quite sparse in practice, which further increases the complexity of latency prediction.

*1) **Matrix Factorization Model**:* Matrix factorization is a classic model to locate the latent factors for prediction. To address this problem, matrix factorization model is widely used [12], [13]. The intuition of this idea is that close users or components will have the similar network condition, thus experience similar latencies to others. Factorizing a matrix is to map both users and components to a joint latent factor space of a low dimensionality d, such that user-component latencies and component-component latencies can be captured as inner products in that space. Towards this end, we use latent feature vectors $U \in \mathbb{R}^{d\times n}$ and $S \in \mathbb{R}^{d\times m}$ to fit L^U, while using $V \in \mathbb{R}^{d\times m}$ and S to fit L^S. To minimize the fitting error, we derive the following model:

$$\begin{aligned}\textbf{min } \Psi(U,S,V) = &\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{m} I_{ij}^{U}(L_{ij}^{U} - U_i' S_j)^2 \\ &+ \frac{1}{2}\sum_{k=1}^{m}\sum_{h=1}^{m} I_{kh}^{S}(L_{kh}^{S} - V_k' S_h)^2 \\ &+ \frac{\lambda_U}{2}\|U\|_F^2 + \frac{\lambda_S}{2}\|S\|_F^2 + \frac{\lambda_V}{2}\|V\|_F^2,\end{aligned} \tag{1}$$

where I_{ij}^U (I_{kh}^S) is the indicator function that equals 1 if user u_i (component s_k) invoked component s_j (component s_h) and 0 otherwise. $\|\cdot\|_F$ denotes the Frobenius norm, and λ_U, λ_S and λ_V are three parameters. For better understanding, we call U_i and V_k as the outgoing virtual coordinate of user u_i and service s_k, S_j and S_h as the ingoing virtual coordinates of service s_j and s_h. Then the latency can be predicted using the one outgoing virtual coordinate and one ingoing virtual coordinate. For example, $U_i' S_j$ denotes the latency of user u_i invoking service $s_j$the latency of users have no ingoing virtual coordinates since we do not need the latency to them.

The optimization model in Eq. 1 minimizes the sum of squared errors of matrices L^U and L^S with quadratic regularization terms, which avoid the overfitting problem under sparse matrices. Normally, batch gradient descent algorithm can be adopted to reach a local minimum of the objective function. Thus, the virtual coordinates can be updated iteratively by

Algorithm 1: Online Latency Prediction Algorithm

Input: Latency data: L_{ij}^U, L_{kh}^S
Output: The virtual coordinates: U_i, S_j, V_k
Randomly initialize $U \in \mathbb{R}^{d\times n}$, and S, $V \in \mathbb{R}^{d\times m}$;
repeat /* Incremental updating */
  Collect historical latency data;
  if *receive a latency data sample (u_i, s_j, L_{ij}^U)* **then**
    $U_i \leftarrow U_i - \eta((U_i^T S_j - L_{ij}^U)S_j + \lambda_u U_i)$;
    $S_j \leftarrow S_j - \eta((U_i^T S_j - L_{ij}^U)U_i + \lambda_s S_j)$;
  else if *receive a latency data sample (s_k, s_h, L_{kh}^S)* **then**
    $S_h \leftarrow S_h - \eta((V_k^T S_h - L_{kh}^S)S_h + \lambda_v V_k)$;
    $V_k \leftarrow V_k - \eta((V_k^T S_h - L_{kh}^S)V_k + \lambda_s S_h)$;
until *converge*;

$$U_i \leftarrow U_i - \eta\frac{\partial\Psi}{\partial U_i},\ \ S_j \leftarrow S_j - \eta\frac{\partial\Psi}{\partial S_j},\ \ V_k \leftarrow V_k - \eta\frac{\partial\Psi}{\partial V_k} \tag{2}$$

where η is the learning rate. However, to adapt to the changes over time, the batch gradient descent algorithm needs to repeat training the whole model even when only one data sample is being updated, which is inefficient.

*2) **Incremental Updating of Virtual Coordinates**:* To overcome this limit, in this paper, we introduce the stochastic gradient descent (SGD) algorithm to our matrix factorization model. SGD is a variant of batch gradient descent, which is often used for online learning [14]. Instead of collecting all the training data and moving on the average gradient descent, each iteration of SGD chooses one training sample and adjust the model stochastically by only taking into account that data sample. This scheme of SGD is very efficient, especially for stream data.

Suppose the new data sample is (u_i, s_j, L_{ij}^U), the objective related to this particular data in Eq. 1 becomes

$$\psi(U_i, S_j) = \frac{1}{2}(L_{ij}^U - U_i' S_j)^2 + \frac{\lambda_u}{2}\|U_i\|_2^2 + \frac{\lambda_s}{2}\|S_j\|_2^2, \tag{3}$$

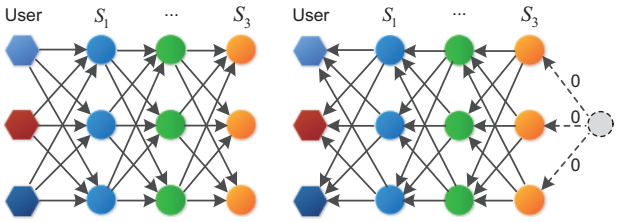
where the first term is the squared error between the real value and predicted value, and the following two terms are the corresponding regularizations. Note that the regularization parameters λ_u and λ_s are on different scale from those in Eq. 1.

Similarly, when receiving data sample (s_k, s_h, L_{kh}^S), the objective value is

$$\psi(V_k, S_h) = \frac{1}{2}(L_{kh}^S - V_k' S_h)^2 + \frac{\lambda_s}{2}\|S_h\|_2^2 + \frac{\lambda_v}{2}\|V_k\|_2^2. \tag{4}$$

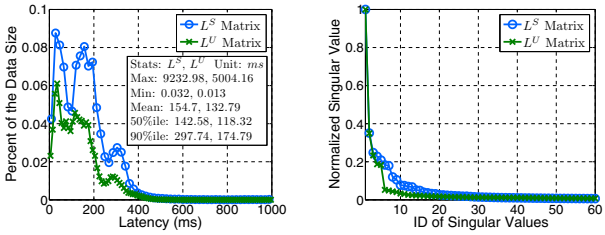
By replacing Ψ with ψ in Eq. 2, we can derive the updating equations to update the virtual coordinates by computing the gradient descent for each objective, $\psi(U_i, S_j)$ and $\psi(V_k, S_h)$. The detailed online latency prediction algorithm is shown in Algorithm 1.

Algorithm 1 is online in the sense that with the stream of latency data, every time we receive a new invocation data sample, we adjust the virtual coordinates by accommodating it to that particular data. And this algorithm is performed continuously to be adaptive to the time varying latency. We do



(a) Invocation Graph (b) Virtual Graph

Fig. 6. Graph Construction



(a) Data Statistics (b) Singular Values

Fig. 7. Data Characteristic

Algorithm 2: Adaptive Component Selection

Input: Critical Path, Virtual coordinates: U_i, S_j, V_k
Output: Component selection strategy

```
1  Construct the virtual graph VG based on the critical path;
2  Topologically sort VG to VG_list;
3  foreach node v in VG_list do /* Initialization */
4      if v ∈ user then
5          v.out ← U_i; v.in ← none;
6      else v.out ← V_k; v.in ← S_j;
7      if v is in the last level of the critical path then
8          v.latency ← 0;
9      else v.latency ← inf;
10     v.parent ← none;
11 foreach node v in VG_list do
12     foreach node w in adjacency of v do
13         if w.latency lg v.latency + w.out'*v.in then
14             w.latency ← v.latency + w.out'*v.in;
15             w.parent ← v;
16 foreach node v ∈ user do /* Output the selection
   strategy v.path */
17     add v.parent to v.path;
```

not provide the convergence of SGD for matrix factorization here, as its detailed proof can be referred to [15].

B. Phase 2: Adaptive Component Selection

1) ***Problem Formulation****:* Given a critical invocation path, we can construct an invocation graph based on the redundant components for each task. Fig. 6(a) illustrate an invocation graph example of the critical path in Fig. 2. Notice that the number of redundant components for each task may be different.

To tolerate the high latency variability and improve the user experience, it is vital to make optimal component selection to minimize the application latency in each time slice and periodically re-optimize the selection strategy. The problem is to find the shortest path in an invocation graph from the user to the last level of components of the critical path. As the shortest path experiences the minimal latency, the components in the shortest path are the optimal request routing strategy.

Conventionally, for example in service computing, only one user is considered for each service composition [16]. However, with the prevalence of cloud computing, the application frontend servers are usually deployed across many locations to serve end users around the world. Hence, there are many users in the invocation graph. The approaches in the literature do not suffice to deal with this problem efficiently, since they make component selection for each user independently. In this way, the complexity will be n times as a single run for one user, which is quite inefficient.

2) ***Shortest Path Algorithm****:* To address this problem, we propose an efficient algorithm, which make the component selection collaboratively for all users by taking full advantage of the structure of invocation graph. To avoid finding the shortest path from each user, we construct a virtual graph based on the original invocation graph, as shown in Fig. 6(b). In the virtual graph, we reverse the edge direction while keeping the original weight. Then a virtual node is added as the source node, thus we can employ the single-source shortest path algorithm. The objective is to find the shortest path from the virtual node to all the user nodes. In this way, we can collaboratively find all the needed shortest paths, which only needs one-time updating.

The detailed algorithm is shown in Algorithm 2. Our algorithm takes advantage of the property that the virtual graph is a directed acyclic graph (DAG), where linear-time shortest path algorithm exists [17].

V. Experiments

A. Data Description

In our experiment, we use three datasets: two real-world latency datasets and one synthetic dataset.

1) ***Dataset1****:* This dataset[1] contains a 1350×460 user-to-component latency matrix and a 460×460 component-to-component latency matrix.

2) ***Dataset2****:* This dataset is extracted from a time-aware Web service QoS dataset[2], which contains the latency data from 4,532 users and 142 components for 64 time slices [18].

3) ***Dataset3****:* : The first two datasets are used to evaluate the accuracy of DR[2]. However, the scale of components is small. To evaluate the efficiency of our approach on different scales, we also randomly generate a latency dataset.

To further capture the characteristics of our dataset, we plot the latency data distribution of the *dataset1* in Fig. 7(a). We can see most of the data is within 200ms. More detailed statistics is provided in the figure for both L^U and L^S. In Fig. 7(b) we plot

[1]http://appsrv.cse.cuhk.edu.hk/~jmzhu/dataset.html
[2]http://www.wsdream.net

TABLE I. PERFORMANCE COMPARISON

Methods	Density = 10%	Density = 20%	Density = 30%	Density = 40%	Density = 50%	Density = 100%
	$ARE \pm std$	$ARE \pm std$	$ARE \pm std$	$ARE \pm std$	$ARE \pm std$	ARE
Random	6.444 ± 0.088	6.446 ± 0.047	6.435 ± 0.043	6.431 ± 0.035	6.405 ± 0.069	6.436
Greedy-M	0.888 ± 0.194	0.613 ± 0.086	0.517 ± 0.110	0.506 ± 0.087	0.496 ± 0.092	0.656
DR^2	**0.412 ± 0.108**	**0.269 ± 0.045**	**0.163 ± 0.043**	**0.129 ± 0.020**	**0.089 ± 0.025**	**0**

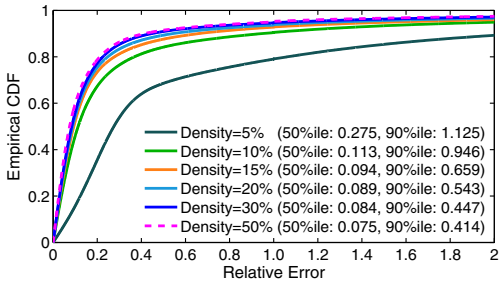


Fig. 8. Prediction Accuracy

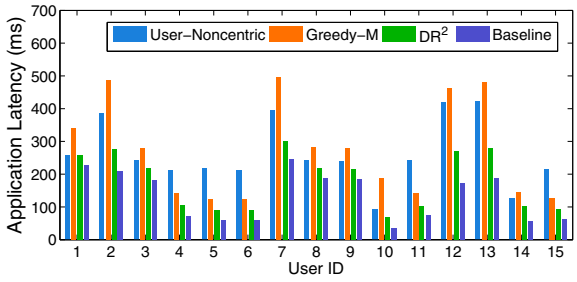


Fig. 9. Performance on Multiple Users

the singular values of both matrices. The singular values are normalized so that the largest singular values of both matrices are equal to 1. We can see that the except the first few large singular values, most of them are close to zero. This means both matrices are approximately low-rank, which conforms to the intuition of matrix factorization model.

B. Performance Evaluation

1) ***Accuracy of Online Latency Prediction****:* To evaluate the prediction accuracy, we conduct the experiments on the *dataset1*. To simulate the real-world sparse data matrix, we randomly remove some of the data entries and remain the data with different densities. Then we predict the missing values using our proposed approach. Fig. 8 shows the cumulative distribution of the relative error. With the increase of data density, the prediction accuracy first increases dramatically, and then the increase diminishes when density $\geq 20\%$.

2) ***Performance Comparison of Component Selection****:* To evaluate the performance of component selection, we compare our method with some other approaches in the following:

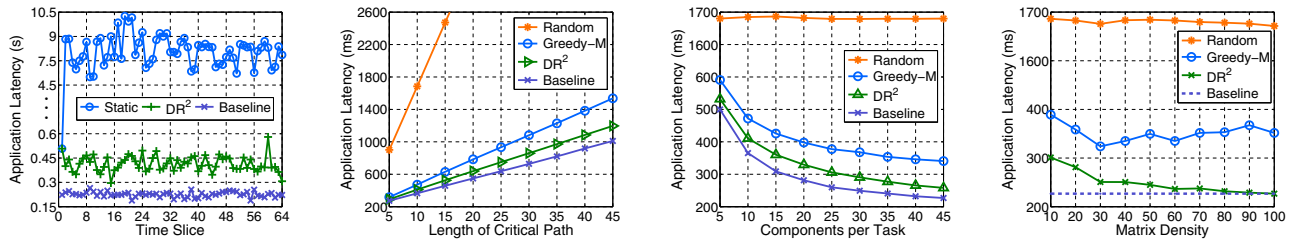
- **Random:** This approach is proposed as a baseline, in which the components are selected randomly from the redundant candidates.
- **Greedy-M:** The greedy approach is to route each user request to the closest component at each step. If there are n users, this greedy algorithm must be run for n times to get request strategy for every user, thus is denoted as Greedy-M. (M for multiple)
- **Dijkstra-M:** This is the most classic algorithm to find the shortest path, which is also used in [11] as a baseline approach. Given a original invocation graph, this method also needs to run multiple times if there are multiple users. We denote it as Dijkstra-M.

To compare the performance, we use the average relative error (ARE) metric. Relative error is the predicted error of the application latency divided by the true latency, while ARE is the average over 100 randomly generated critical paths. For each method, we use the same 100 critical paths.

In this experiment, we set the length of critical path to 10, and each task has 45 components. The results with different densities are shown in Table I, while both the average and the stand deviation are reported, since each experiment runs 20 times for each density. Our performance significantly outperforms the others. As Dijkstra-M has the same accuracy with DR^2, it is omitted for report here. With the increasing of density, the improvement of the performance diminishes, especially when density $\geq 30\%$. Density $= 100\%$ is also listed out since it is the result on the exact pairwise latency. Our method is totally correct. This column has no std since each run has the same result.

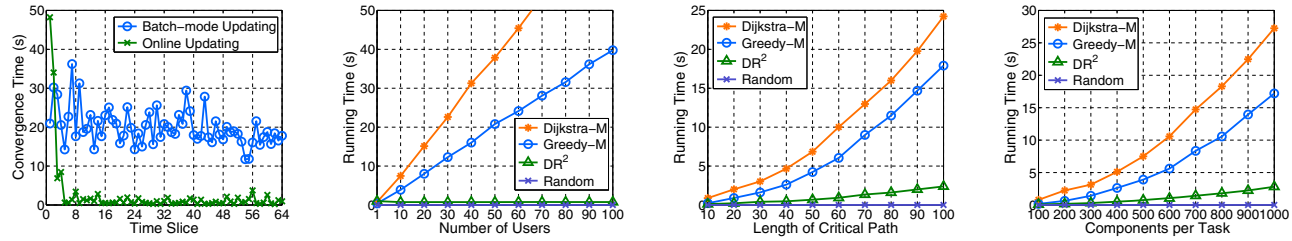
3) ***Performance on Multiple Users****:* A key feature of dynamic request routing is that there are usually multiple users, and user-centric request routing is in demand. In this experiment, we randomly select 15 users and get the average application latency. *User-Noncentric* is the method that does not consider the user-centric property and make the same request routing for each user. Here it uses the same strategy with DR^2 for user 1. *Baseline* employs the exact pairwise latency for component selection, and provides a lower bound. The results are shown in Fig. 9. We can observe that the application latency has high variability over users. And our method DR^2 can be user-centric and obtains good performance, while User-Noncentric and Greedy-M experiences high latency.

4) ***Performance on Multiple Time Slices****:* To evaluate the performance for tolerating the latency variability, experiments are conducted on *dataset2*. *Static* means the static request routing strategy for all time slices. Here it uses the same routing strategy with the one of DR^2 at time slice 1. From Fig. 10(a), we can see that our approach can nearly adapt to the changes and keep consistently low latency, while the static request routing always experience high latency.



(a) Peformance on Multi. Time Slices (b) Impact: Length of Critical Path (c) Impact: Components per Task (d) Impact: Matrix Density

Fig. 10. Performance Evaluation



(a) Convergence Time of DR^2 (b) Running Time v.s. Num. of Users (c) Running Time v.s. Crit. Path Leng. (d) Running Time v.s. Compon. Num.

Fig. 11. Efficiency Evaluation

C. Impact of the Parameters

1) ***Impact of the Length of Critical Path****:* To evaluate the impact of the length of critical path, we vary it from 5 to 45 at a step of 5. In this experiment, we set the number of components to 10, the data density to 30%. From the result shown in Fig. 10(b), we can see that the application latency is linear with the length of critical path.

2) ***Impact of the Number of Components per Task****:* To evaluate the impact of the number of component per task, we vary it from 5 to 45 at a step of 5. And also we set the number of components per task to 10, and the data density to 30%. We can see from Fig. 10(c) that the application latency decreases as the number of components per task becomes larger, except for the random approach, which does not take advantage of the redundant components.

3) ***Impact of the Matrix Density****:* To present a comprehensive evaluation on the impact of the matrix density, we vary the density from 10% to 100% at the step of 10%. Besides, we set the length of critical path to 10 and the number of components per task to 45. For each density, 20 runs are performed and the average is reported, as shown in Fig. 10(d). We can observe that the application latency drops significantly when density increases and approach to the baseline when density $\geq 80\%$. The baseline is the application latency of routing request on exact latency data (density $= 100\%$). However, the random and greedy-M keep independent with the matrix density.

D. Efficiency Analysis

In addition to evaluate the accuracy performance of our proposed approach, we also perform efficiency analysis and make comparisons over the overhead.

1) ***Convergence Time of Online Latency Prediction****:* We first evaluate the convergence time of our online latency prediction algorithm and traditional batch gradient descent by using the *dataset2*. This dataset contains latency data for 64 time slices at a interval of 15 minutes [18]. We train our matrix factorization model by online updating using sequentially coming data samples, while batch gradient descent algorithm re-train the whole model at each time slice. As shown in Fig. 11(a), despite the long convergence time for the first time slice, our online updating algorithm converges very fast in the following time slices. In contrast, batch-mode updating algorithm is more efficient for one-time training, but it needs to re-train the whole model at each time, resulting in the inefficiency, compared to our online algorithm.

2) ***Running Time v.s. Number of Users****:* A key feature different from conventional Web service selection is that there are multiple users to be considered. To study the efficiency of our approach to deal with the multi-user situation, we vary the number of users from 1 to 100, and evaluate the running time of different approaches. As illustrated in Fig. 11(b), our DR^2 approach almost achieves the same efficiency with the random approach in terms of the number of users. Our approach makes component selection collaboratively for all users, thus resulting high efficiency. In contrast, Dijkstra-M and Greedy-M need to run n times by making component selection independently for each user, thus the running time is nearly linear to the number of users, n.

3) ***Running Time v.s. Length of Critical Path / Number of Components per Task****:* To evaluate the scalability of our algorithm to large scale systems, we evaluate the efficiency in terms of both the length of critical path and the number of components per task. Fig. 11(c) and Fig. 11(d) illustrate the experimental results. We can observe that while the running times of both Dijkstra-M and Greedy-M grow fast, our DR^2 approach can almost achieve linear running time in terms of the length of critical path or the number of components per task. Our approach turns out to be very scalable.

VI. Related Work

Cloud applications are gaining more and more popularity nowadays. However, a big challenge of the application latency variability is still to be addressed. In recent literature, a large body of work has been conducted to study the related research problems.

1) ***Latency Prediction****:* The first challenge is to obtain the real-time latency data. Due to the infeasibility of active measurement, many papers resort to employ historical data to predict the unknown values. There are several types of models: network coordinate (e.g., [19], [20]), collaborative filtering (e.g., [21]) and matrix factorization (e.g., [12], [13]). However, these approaches only focus on improving the prediction accuracy, while this paper employs online matrix factorization model to facilitate adaptive component selection and enhance the scalability of our framework.

2) ***QoS-aware Service Selection****:* Similar to our component selection problem, QoS-aware service selection problem has been widely studied [16], [11], which aims at finding optimal service candidates to optimize the performance of composite service. However, different with the cloud applications which have multiple users, they typically only find one solution, which cannot be used to address our problem.

3) ***Service Adaptation****:* Another related technique is service adaptation in service-oriented systems [22], [23], which can support QoS-driven adaptation to satisfy the performance requirements of users. However, these methods did not consider the dynamic latency problem due to the uncertain network conditions, with the common premise that the QoS of service candidates can be obtained from the service level agreement (SLA), which is impractical in fact.

VII. Conclusion and Future Work

In this paper, to tolerate the latency variability in cloud applications, we consider the dynamic request routing approach, DR^2, by jointly perform online latency prediction and adaptive component selection. Towards this end, online matrix factorization is proposed to predict the real-time latency data, and an efficient shortest path algorithm is utilized to make component selection. Extensive experiments are conducted and the results show the effectiveness and efficiency of our approach.

In this paper, we only focus on the latency minimization and variability tolerating problem, while the load balancing among the redundant components is not considered. As a meaningful future work, we will explore more to jointly consider the application latency minimization and load balancing problem.

Acknowledgment

The work described in this paper was fully supported by the National Basic Research Program of China (973 Project No. 2011CB302603), the National Natural Science Foundation of China (Project No. 61100078), the Shenzhen Basic Research Program (Project No. JCYJ20120619153834216, JC201104220300A), and the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK 415212).

References

[1] X. Lu, H. Wang, J. Wang, J. Xu, and D. Li, "Internet-based virtual computing environment: Beyond the data center as a computer," *Future Generation Comp. Syst.*, vol. 29, no. 1, pp. 309–322, 2013.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[3] D. Strom and J. F. van der Zwet, "Truth and lies about latency in the cloud," Interxion^{TM} white paper, 2012.

[4] Y. Yoon, C. Ye, and H.-A. Jacobsen, "A distributed framework for reliable and efficient service choreographies," in *Proc. of ACM WWW*, 2011, pp. 785–794.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proc. of ACM SOSP*, 2007, pp. 205–220.

[6] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[7] Q. Zhang, Q. Zhu, M. F. Zhani, and R. Boutaba, "Dynamic service placement in geographically distributed clouds," in *Proc. of IEEE ICDCS*, 2012, pp. 526–535.

[8] M. Alicherry and T. V. Lakshman, "Network aware resource allocation in distributed clouds," in *Proc. of IEEE INFOCOM*, 2012, pp. 963–971.

[9] J. Zhu, Z. Zheng, Y. Zhou, and M. R. Lyu, "Scaling service-oriented applications into geo-distributed clouds," in *Proc. of the International Workshop on Internet-based Virtual Computing Environment (iVCE)*, 2013.

[10] J. Dai, Z. Hu, B. Li, J. Liu, and B. Li, "Collaborative hierarchical caching with dynamic request routing for massive content distribution," in *Proc. of IEEE INFOCOM*, 2012, pp. 2444–2452.

[11] A. Klein, F. Ishikawa, and S. Honiden, "Towards network-aware service composition in the cloud," in *Proc. of ACM WWW*, 2012, pp. 959–968.

[12] Y. Liao, P. Geurts, and G. Leduc, "Network distance prediction based on decentralized matrix factorization," in *Proc. of IFIP Networking*, 2010, pp. 15–26.

[13] W. Lo, J. Yin, S. Deng, Y. Li, and Z. Wu, "An extended matrix factorization approach for QoS prediction in service selection," in *Proc. of IEEE SCC*, 2012, pp. 162–169.

[14] G. Ling, H. Yang, I. King, and M. R. Lyu, "Online learning for collaborative filtering," in *Proc. of the International Joint Conference on Neural Networks (IJCNN)*, 2012, pp. 1–8.

[15] A. Shapiro and Y. Wardi, "Convergence analysis of gradient descent stochastic algorithms," *Journal of Optimization Theory and Applications*, vol. 91, pp. 439–454, 1996.

[16] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Trans. Software Eng.*, vol. 33, no. 6, 2007.

[17] R. L. R. T. H. Cormen, C. E. Leiserson and C. Stein, *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[18] Y. Zhang, Z. Zheng, and M. R. Lyu, "Exploring latent features for memory-based QoS prediction in cloud computing," in *Proc. of IEEE SRDS*, 2011, pp. 1–10.

[19] T. S. E. Ng and H. Zhang, "Predicting internet network distance with coordinates-based approaches," in *Proc. of IEEE INFOCOM*, 2002.

[20] J. Zhu, Y. Kang, Z. Zheng, and M. R. Lyu, "WSP: A network coordinate based web service positioning framework for response time prediction," in *Proc. of IEEE ICWS*, 2012, pp. 90–97.

[21] Z. Zheng, H. Ma, M. R. Lyu, and I. King, "WSRec: A collaborative filtering based web service recommender system," in *Proc. of the IEEE ICWS*, 2009, pp. 437–444.

[22] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, and R. Mirandola, "MOSES: A framework for QoS driven runtime adaptation of service-oriented systems," *IEEE Trans. Software Eng.*, vol. 38, no. 5, pp. 1138–1159, 2012.

[23] V. Nallur and R. Bahsoon, "A decentralized self-adaptation mechanism for service-based applications in the cloud," *IEEE Trans. Software Eng.*, preprint.