Combining Quality of Service and Social Information for Ranking Services

Qinyi Wu[#], Arun Iyengar[*], Revathi Subramanian[*], Isabelle Rouvellou[*], Ignacio Silva-Lepe[*], Thomas Mikalsen[*]

[#] *College of Computing, Georgia Institute of Technology*
801 Atlantic Drive, Atlanta, GA 30332, USA
qxw@cc.gatech.edu

[*] *IBM T.J. Watson Research Center*
19 Skyline Drive, Hawthorne, NY 10532, USA
{*aruni, revathi, rouvellou, isilval, tommi*}*@us.ibm.com*

Abstract. In service-oriented computing, multiple services often exist to perform similar functions. In these situations, it is essential to have good ways for qualitatively ranking the services. In this paper, we present a new ranking method, ServiceRank, which considers quality of service aspects (such as response time and availability) as well as social perspectives of services (such as how they invoke each other via service composition). With this new ranking method, a service which provides good quality of service and is invoked more frequently by others is more trusted by the community and will be assigned a higher rank. ServiceRank has been implemented on SOAlive, a platform for creating and managing services and situational applications. We present experimental results which show noticeable differences between the quality of service of commonly used mapping services on the Web. We also demonstrate properties of ServiceRank by simulated experiments and analyze its performance on SOAlive.

Key words: Cloud computing, Quality of service, Service ranking

1 Introduction

Cloud computing is viewed as a major logical step in the evolution of the Internet as a source of services. With many big companies now offering hosted infrastructure tools and services, more and more businesses are using cloud computing. We envision an open, collaborative ecosystem where cloud services can be easily advertised, discovered, composed and deployed. In cloud computing, there are often software services that perform comparable functions. An example would be mapping services such as those available from Google, Yahoo!, and Mapquest. Users, service composers and service invokers alike are thus faced with the task of picking from a set of comparable services that meet their needs. A random selection may not be optimal for its targeted execution environment and may incur inefficiencies and costs. In this situation, it will be very valuable if users could be provided with some indication of the relative merits of comparable services. We propose a new ranking method to address this need.

Our methodology takes into account how services invoke each other via service composition. Service composition allows developers to quickly build new applications using existing services that provide a subset of the function they need. An address book service that takes as input an address and returns its geocoding is an example of a primitive service that provides a specialized function. A FindRoute service that takes as input the geocoding of two addresses and returns a route from the start address to the end address is a composite service. Composite services can also become the building blocks of other composite services. The ability to compose and deploy services quickly is a big draw for existing and prospective cloud customers. We therefore imagine that cloud environments shall abound in service networks, where services form client-server relationships. Having good methodologies for evaluating and ranking services will be critically important for selecting the right services in this environment.

Our service ranking methodology incorporates features from social computing. Social ranking features are available throughout the Web. The social rank of an item is the popularity of the item within a community of users. This community can be virtual or real. Recommender systems such as those by Amazon or Netflix collect reviews and ratings from users and record their preferences. They can then use this information to recommend products to like-minded users (a virtual social network of users). Social bookmarking sites such as del.icio.us allow an explicit community of users to be formed via user networks. del.icio.us provides listings of the most popular bookmarks at any point in time which can be tailored to specific communities. There has also been past work in ranking and matching web services [1][13]. Prior research deals with finding the services that best match a required interface, support certain functions, or satisfy certain rules or policies.

In our approach to rank services in the cloud, we start out with the assumption that some initial matchmaking has been performed to arrive at a set of comparable services which then need to be ranked. We therefore, do not dwell on the aspects of matching interfaces, service descriptions, semantics, etc. Instead, we focus our energies on drawing the parallel between social networks and service networks. In social networks, users rate services. In service networks, services can rate other services based on how successful the service invocations were. The high rank (or popularity) of a service is influenced not just by a large number of service clients, but also by the satisfaction expressed by these service clients. A hike in the rank of a service S propagates favorably down the line to other services that it (S) depends upon.

The contributions of this paper are as follows:

- We present a new algorithm, referred to as ServiceRank, for ranking services which combines quality of service (QoS) aspects such as response time and availability with social ranking aspects such as how frequently the service is invoked by others.

– We show through experimental results that our algorithm is efficient and consumes minimal overhead.
– We study the performance of different mapping services on the Web. Our results indicate that the different services exhibit different behavior which is a key reason that quantitative methods are needed to rank services.

In the rest of this paper, we first define the ServiceRank algorithm in Section 2. We then describe an implementation of ServiceRank in Section 3. Experiments are presented in Section 4. We describe related work in Section 5 and conclude in Section 6.

2 ServiceRank

ServiceRank incorporates features from social computing by taking into account how services invoke each other via service composition. Figure 1 shows an example. A circle represents a service. A directional arrow represents a service invoking another service to fulfill its functionality. We call the service sending a request the *client* and the service processing the request the *server*. In this example, services s_1 and s_2 are clients. s_4 is their server. s_3 dynamically invokes either s_4 or s_5 to balance its load between these two services. s_4 and s_5 are grouped into a category because they provide the same functionality. From ServiceRank's perspective, a request is regarded as a rating from the client to the server. The client evaluates all the requests to compute a local rating of the server. Local ratings are eventually aggregated to compute global ranks for all the services. ServiceRank considers three factors for the aggregation. The first factor is how many clients a service has. In this example, we expect that s_4 gains a higher rank than s_5 because it has more client services. The second factor is how frequently a service is invoked. If s_3 sends more requests to s_5 than s_4, s_3 will rank s_5 higher under the condition that the quality of both services is similar. The third factor considers QoS in terms of response time and availability. For example, if s_5 has better response time than s_4, its rank should be raised even though it has fewer clients. In the rest of this section, we explain how ServiceRank combines all three factors to compute global ranks for services.

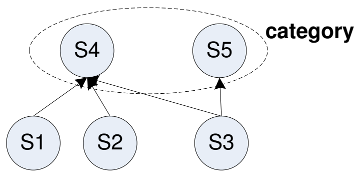


Fig. 1. A service network example

2.1 Local Ratings

A service network consists of a set of services $S = s_1, s_2, ..., s_n$. If s_i sends a request to s_j, s_i is a *client* of s_j, and s_j a *server* of s_i. We use $R_{ij} = \{r_{ij}^1, r_{ij}^2, ..., r_{ij}^m\}$ to denote all the requests between s_i and s_j and r_{ij}^u the u-th request. In ServiceRank, a request r_{ij}^u is regarded as a rating from s_i to s_j. If s_i processes it successfully, s_i gives s_j a positive rating: $rate(r_{ij}^u) = 1$, otherwise $rate(r_{ij}^u) = -1$. s_i's total rating to s_j, denoted by l_{ij}, is the sum of the ratings of all the requests.

$$l_{ij} = \sum_u rate(r_{ij}^u) \tag{1}$$

l_{ij} considers how frequently a service is invoked, and whether requests are successfully processed. However, QoS is a critical factor in service composition as well. It is important that ranks of services can be differentiated based on their performance. The ServiceRank algorithm achieves this goal by comparing the average response time of a service with that of other services with the same functionality and using the comparison ratio to adjust local ratings. Next we introduce a few more notations to explain how this is done.

For a service s_j, its average response time, rt_j, is computed by averaging the response time of all the requests it receives. Let B_j denote its client set.

$$rt_j = \frac{\sum_{s_i \in B_j} \sum_{r_{ij}^u \in R_{ij}} response\ time(r_{ij}^u)}{\sum_{s_i \in B_j} |R_{ij}|} \tag{2}$$

Services with the same functionality are grouped into a category, denoted by c_u. We use $min\ c_u^{rt}$ to denote the minimal average response time of services in c_u. Services with no requests are not considered. Suppose s_j belongs to c_u, the total rating from s_i to s_j is adjusted as follows:

$$\hat{l_{ij}} = (\sum_u rate(r_{ij}^u)) * \frac{min\ c_u^{rt}}{rt_j} \tag{3}$$

In the above equation, if s_j achieves the minimal average response time in category c_u, the total rating remains the same. Otherwise, the rating will be adjusted by a constant less than 1. $\frac{min\ c_u^{rt}}{rt_j}$ brings category knowledge into local ratings. This unique feature differentiates ServiceRank from earlier ranking algorithms in which local ratings are solely based on local knowledge [5][7]. With this new feature, if a client sends the same amount of requests to two services in a category, the client will give a higher rating to the one achieving better response time. Note that there are other ways to adjusting local ratings with category knowledge. For example, we can use the median of services' average response time and put penalties on local ratings only when a service performs below the average. We do not discuss them further since they do not change the definition of Equation 3 fundamentally. A concern in this approach is that a malicious service can register itself in a category and respond back to its malicious partners instantaneously. In doing so, an ordinary service is likely to be penalized due to its "bad" performance. This problem can be avoided if we use average response time from well-established services as an adjusting baseline.

2.2 Normalizing and Aggregating Local Ratings

In social ranking, we wish that the rank of a service is decided by both the ranks and ratings of its clients. ServiceRank computes the global rank of s_j by aggregating the local ratings from its client, defined as

$$w_j = \sum_{s_i \in B_j} \hat{l_{ij}} w_i \tag{4}$$

It is important to normalize local ratings to remove noisy data and protect the ranking system from a malicious party which creates bogus services and commands them to send requests to a service to artificially raise its rank. ServiceRank normalizes local ratings in two steps. First, it evaluates the eligibility of a local rating l_{ij} by two criteria: 1) the total number of requests exceeds a constant number T such that $|R_{ij}| > T$; 2) successful rate exceeds a threshold β such that $\frac{|R_{ij}^{succ}|}{|R_{ij}|} > \beta$, where R_{ij}^{succ} denotes those requests that satisfy $rate(r_{ij}^{u}) = 1$. T and β are two configurable parameters. The two criteria ensure that two services must establish a stable history before ServiceRank considers its local rating. This helps remove noisy data such as ratings from testing requests or ratings for unavailable services. In the second normalization step, only eligible ratings are considered. A local rating from s_i to s_j is divided by l_{ij} with the total number of requests sent by s_i:

$$r_{ij} = \frac{\hat{l_{ij}}}{\sum_j \hat{l_{ij}}} \tag{5}$$

With Equation 4, the global rank values of services $w = (w_1, w_2, ..., w_n)$ are the entries of the principal left eigenvector of the normalized local rating matrix $R = (r_{ij})_{ij}$, defined as follows:

$$w^T = w^T R \tag{6}$$

The above definition does not consider prior knowledge of popular services. In a service network, some services are known to be trustworthy and provide good quality. Similar to the early approach [5], ServiceRank uses this knowledge to address the problem of malicious collectives in which a group of services sends requests to each other to gain high global ranking values. Let Q denote a set of trusted services. We define the vector q to represent the pre-trusted rank values. q_i is assigned a positive value if $s_i \in Q$, otherwsie $q_i = 0$. q satisfies $\sum_i q_i = 1$. The global rank values of services are now defined as:

$$w^T = a w^T R + (1 - a) q^T \tag{7}$$

where R is the normalized local rating matrix and a a constant less than 1. Equation 7 is a flow model [4]. It assumes that the sum of global ranks is a constant, and the total rank value is distributed among services in a network. q serves as a rank source. This model states that starting from a trusted source, the global ranks are unlikely to be distributed to untrusted services if there are no links from trusted services to untrusted services. Therefore, malicious collectives can be prevented if we can effectively control the number of links between these two groups.

3 System Prototype and Runtime traffic monitoring

ServiceRank has been implemented on SOAlive [12] which is an approach to provide smart middleware as a service. The SOAlive platform allows users to create, deploy, and manage situational applications easily. Each application may include one or more services which may be invoked at runtime, and which in turn, may invoke other services. For our experiments, we used a SOAlive implementation on WebSphere sMash [http://www.projectzero.org/]. WebSphere sMash is an agile web development platform that provides a new programming model and a runtime that promotes REST-centric application architectures. Logically, the SOAlive platform can be broken down into i) system components; and ii) hosted applications and their runtimes.

Figure 2 shows the key SOAlive components. The service catalog, the repository, the application manager, the application installer, and the router work in concert to provide a simplified development and deployment experience.

- The **SOAlive repository** allows modules, the building blocks for applications, to be uploaded and shared.
- The **SOAlive Application Manager** lets users create deployed applications from deployable modules in the repository.
- The **SOAlive Application Installer** is responsible for downloading, resolving, and installing user applications on worker nodes.
- The **SOAlive catalog** stores metadata about hosted artifacts, in addition to storing metadata about external artifacts which are of interest to users of the SOALive platform. It also acts as the hub for collaborative development.
- The **SOAlive router** is the first stop for any request coming into SOAlive and provides a suitable extension point for monitoring functions.

SOAlive supports several different topologies ranging from the one in which all the system components and managed applications run on a single node to a truly distributed topology where individual system components are themselves distributed across several nodes, and applications execute in one or more worker nodes based on system policies. SOAlive defines several extension points as a way to build upon its core functionality. One of these extension points allows for different runtime monitors to be added as logical extensions to the routing component.

Monitoring is enabled on a per-application basis. Each application includes a “monitor” flag that must be set for monitoring to occur. When monitoring is enabled for a given application, the server will invoke all registered application monitors for each application request/response pair. The monitor will be invoked on the application request thread, after the response is available. The monitor’s

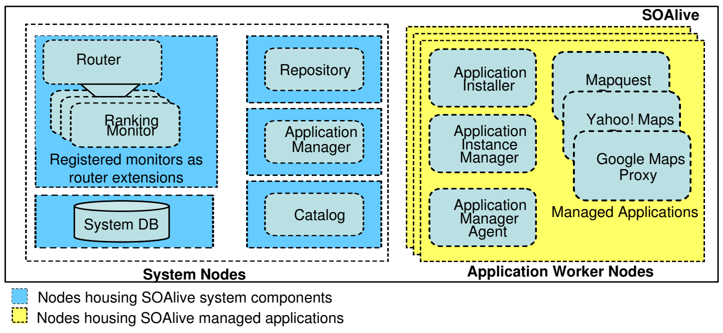


Fig. 2. SOAlive system architecture

caller assumes that the monitor will return as quickly as possible, and that it will defer any processing for a later time and on a separate thread. The following figure shows the sequence of events when SOAlive receives a request to a managed application that has monitoring enabled. For inter-application requests (i.e., where one hosted SOAlive application invokes another hosted SOAlive application), the runtime for the source application adds headers to the out-bound request that identifies the source application and the specific method in the source application from which the call originated. This allows the SOAlive monitoring and logging facilities to fully determine the source of a request. This header injection feature is also used to propagate the correlator for a chain of invocations. For instance, if application A_1 called A_2 that called A_3 and A_4, then the paths $A_1 \to A_2 \to A_3$ and $A_1 \to A_2 \to A_4$ have the same correlator. This correlator is a unique ID generated by SOAlive at the start of a chain of requests.

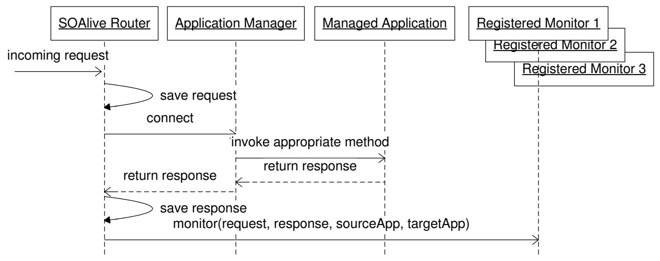


Fig. 3. SOAlive monitoring flow

Using the information gathered from the ranking monitors, ranks are computed periodically and incrementally in SOAlive. Weights are assigned to each evaluation, with the more recent evaluations having higher weights. The assigned weights also depend upon service lifecycle events - for example, if a service is entirely rewritten, then its previous evaluations are assigned low weights. If a minor bug fix is made to a service, then the earlier ratings still have considerable importance, and therefore higher weights. Our incremental ranking procedure allows new evaluations to update ranks without the need to re-examine old evaluations.

4 Experiment Results

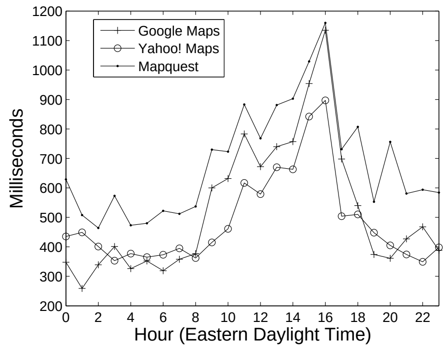
4.1 Map Services

ServiceRank is designed to take QoS into consideration because we expect services demonstrate dynamic behavior and should be ranked differently. We conducted experiments on real-world services to confirm this expectation. In our experiments, we collected traffic data from three well-known map services: Google Maps, Yahoo! Maps, and Mapquest. They were chosen because all of them have standard APIs that take the geocoding of an address and return its local map. Moreover, the returned results all contain similar map data. Therefore, it is meaningful to characterize and compare their performance in terms of response time and failure rate.

Experiment setup To obtain the traffic data of three map services through SOALive, we create three proxies. Each proxy is responsible for forwarding a request to the real map service and forwarding back the result to its client. We implement a workload generator that periodically sent requests to three proxies at a configurable interval. At each turn, the workload generator uniformly chooses the geocoding of an address in the US from a database that contains hundreds of entries. We collect the traffic data for each service for seven consecutive days. The time interval is set to be 30 seconds. The traffic data was collected from 7:00pm (EDT) August 4th, 2008 to 7:00pm (EDT) August 11th, 2008.

Experiment results Figure 4 shows the average response time at different times in the day. We can see three phenomena. First, all three map services have degraded response time during peak hours (approximately between 8:00 and 18:00). Second, MapQuest has slightly worse response times in general compared to the other two map services. Third, even though Google Maps and Yahoo! Maps have similar response time during non-peak hours, Google Maps performs worse than Yahoo! Maps during peak hours. Figure 5 shows the percentage of failed invocations at different times in the day. The figure does not show anything for Yahoo! Maps because it did not return any failed invocations during our experimental period. Both Google Maps and MapQuest have very small failure rates with MapQuest being slightly higher.

From these experiment results, we can see that real-world services do demonstrate different behavior over time. Therefore, it is very important to rank them dynamically to characterize their latest performance. From Figure 4, we see that Google Maps has degraded response time during peak daytime hours. The most likely explanation is that Google Maps is more loaded during that period. Yahoo! Maps demonstrates better average response time during the same period. QoS-based ranking can provide valuable information to assist applications that have



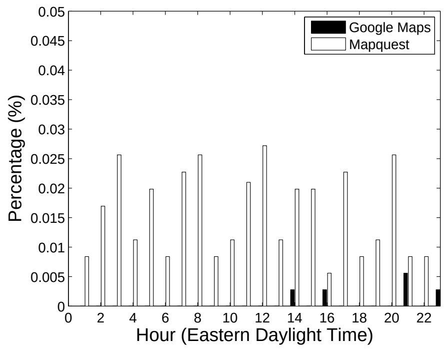


Fig. 4. Average response time

Fig. 5. Percentage of failed invocations. Yahoo! Maps does not have bars because it did not return any failed request. Google Maps has few bars because it returned failed requests only in some of the hours.

critical requirements on performance. For example, travel planning services that want to integrate a map service would do well to choose Yahoo! Maps during peak hours.

The relative performance of Google Maps, Yahoo! Maps, and Mapquest may have changed since the time these measurements were made. For a large number of customers, all three services offer performance and availability which are more than adequate. We do not have sufficient data to judge one of the services as currently being superior to another. The key point is that at any given point in time, different services offering the same functionality will often show noticeable differences in performance. In addition, there may also be considerable variations in performance based on the time of day.

4.2 ServiceRank Properties

We now demonstrate the properties of ServiceRank through a hierarchical service network model. In this model, a set of services form a hierarchical structure. The structure is divided into layers $l_1, l_2, ..., l_n$. Services at the same layer belong to the same category. The lowest layer is l_1. Services at l_i are clients of services at l_{i+1}. Requests are sent by a root service s_0 to services at l_1. To process a request, a service at l_1 invokes one of the services at l_2, which will in turn uses a service at l_3 and so on. The response time of a request at a service is the service's own processing time plus the round trip time spent at upper layers.

Experiment 1 We intend to show how the rank of a service changes with the number of times it is invoked. A simple topology suffices for this purpose. We use the one shown in Figure 6. An arrow represents a client-server relationship. We gradually adjust the percentage of requests between s_1 and s_2 and observe

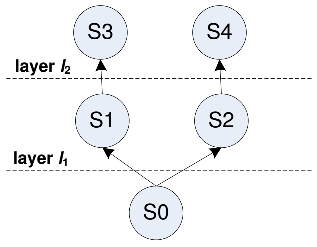
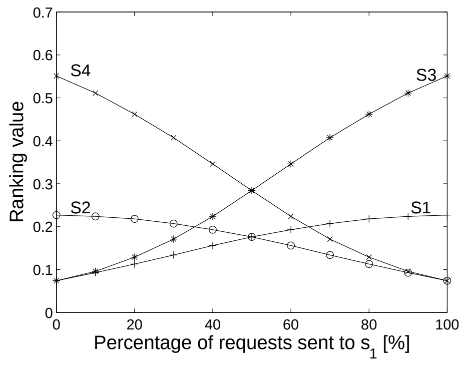


Fig. 6. A simple hierarchical service network



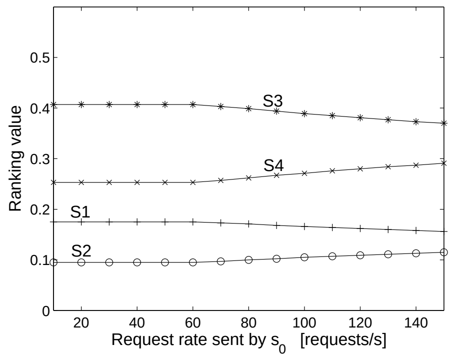


Fig. 7. Impact of request frequency on ranking values

Fig. 8. Impact of response time on ranking values

how the ranks of s_3, s_4, s_5, and s_6 change. To see only the impact of request frequency, we do not consider the two other factors: failure rate and response time. In other words, each request is successfully satisfied, and services in the same category have similar response times. The results are shown in Figure 7. We can see that the ranking values of both s_1 and s_3 increase as they consume a higher percentage of requests compared to their counterparts. The ranking values of s_2 and s_4 decrease correspondingly. We can also see that the ranking value of a service is impacted not only by the percentage of requests it receives, but also by the ranking values of its client. In this case, s_3's ranking value increases faster than s_1 because both s_3's request percentage and s_1's ranking value get increased. This property is a desirable feature of social ranking because it takes into consideration both popularities of services and the amount of workload they share.

Experiment 2 We now evaluate how the rank of a service is impacted by the quality its requests receive. We continue to use the topology in Figure 6. We assume that the percentage of requests between s_1 and s_2 follows the 80-20 rule in which s_1 receives 80% of requests while s_2 receives 20%. The experiment runs in cycles. In each cycle, s_0 sends requests at a given rate to both s_1 and

s_2, which will invoke their corresponding services at the next upper layer. We simulate the average response time of a service by a function, which remains constant when the request rate below a threshold and increases linearly after that. In the experiment, we configure the threshold to be 50. Figure 8 shows the result. Without considering the factor of response time, the ranking values of all services would not change over the course of the experiment because the percentage of requests at all services does not change. After taking response time into consideration, the number of ratings a service receives from its clients will be adjusted by how well the requests are served. From Figure 8, we can see that the ranks of services do not change when the number of requests is below 50. After that, the ranks of s_1 and s_3 begin to drop because their response times start to increase. This is to simulate the situation in which a service shows degraded performance when overloaded. As a result, the ranks of s_1 and s_2 start to converge. s_3 and s_4 demonstrate similar trends. In real applications, this property motivates service writers to improve service response times in order to keep service ranks from declining when the services are overloaded due to high request rates. It also provides more accurate information to guide new traffic to services that are less overloaded.

4.3 Monitoring Overhead in SOALive

SOALive collects the traffic data of services when they are serving customers. It is very important that the monitoring procedure does not interfere with the ordinary operation of services. The experiment in this section measures the monitoring overhead.

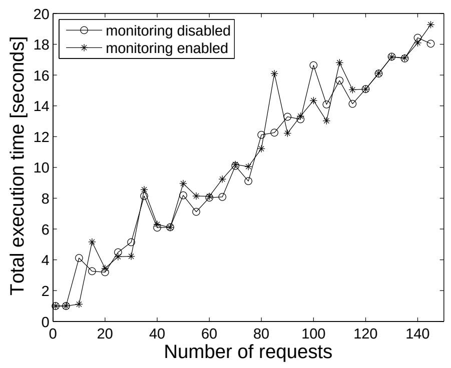


Fig. 9. Monitoring overhead in SOALive

Experiment setup We set up two services s_i and s_j in SOALive. s_i uses s_j's functionality by sending a sequence of HTTP requests. The monitoring service in SOALive is responsible for recording the round trip time of each invocation and its status. s_j has an empty function body. It returns back to s_i as soon as it receives a request. Therefore, the total amount of time to process a batch of requests will be close to the overhead introduced by SOALive.

Experiment result Figure 9 shows the results. We gradually increase the total number of requests between s_i and s_j. For each configuration, we collect total processing time with and without monitoring enabled. We run the experiment five times and compute the average. Figure 9 shows that the overall processing time with monitoring enabled is only slightly higher than the case with monitoring disabled. This demonstrates that an efficient monitoring service can be implemented in a cloud. The current experiment is only run in a small setting. For large settings with hundreds or even thousands of services deployed, we can use different optimization techniques such as sampling to collect traffic data.

4.4 ServiceRank Performance

In SOALive, ServiceRank periodically analyzes traffic data and computes the ranking values of services. It is important that ServiceRank can scale up to large numbers of services to provide ranking values in a timely fashion. We have implemented the ServiceRank algorithm by using the power method to compute the left principal eigenvector of Equation 7. Since we do not have enough services in SOALive to test the algorithm for a large number of services, we evaluate its performance for a high number of services by simulation.

Service network model The topology of a service network is determined by both the number of services and the service invocations. We assume that within the cloud, services with different popularities exist. For an invoked service, the number of its clients conforms to a power law distribution as shown in Table 1. In this setting, 35% of services are only clients and do not provide services to others. A majority of services (60.36%) have clients ranging between 1 and 20. Less than 1% services have more than 100 clients.

Table 1. Distribution of number of clients for invoked services

number of clients	percentage	number of clients	percentage
0	35%	$[21, 40]$	2.26%
$[1, 5]$	48.69%	$[41, 100]$	1.39%
$[6, 20]$	11.67%	$[100, +]$	0.99%

Simulation execution We evaluate the response time and throughput of our ranking algorithm by simulation. In each simulation cycle, a workload generator sends requests at different rates (i.e. the number of requests per second). We use a thread pool to process concurrent requests. Each request computes the ranking values for a service network with a given number of services. The services are connected according to our service network model. To measure average response time, we run our workload generator for three minutes and average the response time of each request. To measure maximum throughput, we adjust

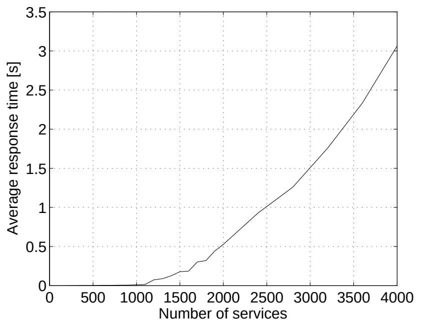


Fig. 10. Average response time for service networks with different number of services

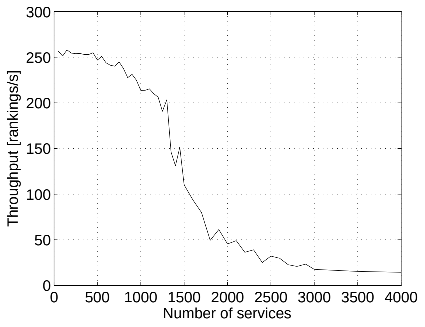


Fig. 11. Throughput per second for service networks with different number of services

request rates and observe the values of throughput at different rates for three minutes. The maximum throughput is the point when the throughput does not increase any more with the increase of request rate.

Hardware configuration All experiments are conducted on a 64-bit GNU Linux machine with Intel(R) Core(TM)2 Quad CPU 2.83GHz, 4GB RAM.

Experiment results Figure 10 shows the results for the measurement of response time. Figure 11 shows the results for the measurement of throughput. As the number of services increases in a service network, the response time is less than tens of milliseconds on average until the number reaches at around 1500 services. Correspondingly, the throughput of ServiceRank scales well for service networks with less than 1500 services. After that the throughput gradually drops from hundreds of rankings per second to less than ten rankings per second. We expect in a real cloud, rankings are not workload-intensive. There may be many seconds between successive rankings. Therefore, ServiceRank should be able to scale up to large settings with many thousands of services.

5 Related Work

Past work addresses the ranking problem by analyzing relationships between different parties. Mei et. al. [7] analyze binding information in service WSDL specifications and apply the PageRank algorithm [9] to compute global ranks of services. The binding relationships are static and cannot distinguish services different in runtime qualities. Gekas et.al. [3] analyze semantic compatibility of input/output parameters of services and select the best matching service for an output request. We focus on QoS metrics for service composition. Two pieces of work are close to ours. One is EigenTrust [5], which works on peer ranking on P2P networks. EigenTrust considers how frequently two parties interact with each other and uses this information to compute global ranks for them. A unique feature of our approach is that we use global knowledge to adjust local ratings to consider the impact of response time. This feature makes our approach better suitable for service ranking in that QoS is a critical factor for service composition. The other related work is [10], which applies document classification techniques for web API categorization and ranks APIs in each category by combining user feedback and utilization. Similar to their work, we also model the service ranking problem by using statistics collected from web traffic. However, [10] considers the factor of popularity only. Our approach additionally considers response time and failure rate and can be easily extended to include user feedback as well.

Other ranking approaches include those based on user feedback or testing techniques. In [2], the authors propose to rank services based on users' ratings to different QoS metrics. These ratings are then aggregated to compute global ranks of services. In [8], gaps between users' feedback and actually delivered QoS from service providers are measured to rank services. These approaches have limited application in service networks because human feedback may not be available for those backend services that do not have direct interactions with customers. Tsai et.al. [14] propose a ranking technique in which pre-developed testing cases are executed periodically to check the current status of services. Services are ranked according to their deviation from the expected output.

Several ranking frameworks are proposed to rank services by combining many aspects of QoS into the same picture. Liu et al. [6] proposed to rank services based on prices, advertised QoS information from service providers, feedback from users, and performance data from live monitoring. Sheth et. al [11] proposed a service-oriented middleware for QoS management by taking into consideration time, cost, reliability and fidelity. Bottaro et al. [1] proposed a context management infrastructure in which services are dynamically ranked based on application contextual states at runtime (e.g., physical location of mobile devices). These frameworks target a broader spectrum of QoS domains and mainly focus on the design of expressive QoS specification languages and algorithmic solutions to aggregating metrics from different subdomains. By comparison, our work provides a unique solution to incorporate QoS into service ranking and can be adopted as part of a broader ranking framework covering other aspects.

6 Conclusion

In cloud computing, services are discovered, selected, and composed to satisfy application requirements. It is often the case that multiple services exist to perform similar functions. To facilitate the selection process for comparable services, we propose a new ranking method, referred to as ServiceRank, that combines quantitative QoS metrics with social aspects of services to provide valuable ranking information. Services form a social network through client-server invocation relationships. The ServiceRank algorithm ranks a service by considering not only its response time and availability but also its popularity in terms of how many services are its clients and how frequently it is used. By combining all these factors, the rank of a service will be raised if it attracts a higher amount of traffic and demonstrates better performance compared to other comparable services. In the future, we plan to integrate service level agreements into our current work. With this feature, the rank of a service will be impacted by both its performance and its fulfillment of service-level contracts.

References

1. André Bottaro and Richard S. Hall. Dynamic contextual service ranking. In *Software Composition*, pages 129–143, 2007.
2. Hoi Chan, Tieu Chieu, and Thomas Kwok. Autonomic ranking and selection of web services by using single value decomposition technique. In *ICWS*, pages 661–666, 2008.
3. John Gekas and Maria Fasli. Automatic web service composition based on graph network analysis metrics. In *OTM Conferences (2)*, pages 1571–1587, 2005.
4. Audun Jósang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decis. Support Syst.*, 43(2):618–644, 2007.
5. Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 640–651, New York, NY, USA, 2003. ACM.
6. Yutu Liu, Anne H. Ngu, and Liang Z. Zeng. Qos computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 66–73, New York, NY, USA, 2004. ACM.
7. Lijun Mei, W. K. Chan, and T. H. Tse. An adaptive service selection approach to service composition. In *Proceedings of the 2008 IEEE International Conference on Web Services*, pages 70–77, Washington, DC, USA, 2008. IEEE Computer Society.
8. Mourad Ouzzani and Athman Bouguettaya. Efficient access to web services. *IEEE Internet Computing*, 8(2):34–44, 2004.
9. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.
10. Ajith Ranabahu, Meenakshi Nagarajan, Amit P. Sheth, and Kunal Verma. A faceted classification based approach to search and rank web apis. In *Proceedings of ICWS'08*, pages 177–184, 2008.
11. A. Sheth, J. Cardoso, J. Miller, and K. Kochut. Qos for service-oriented middleware. In *Proceedings of the Conference on Systemics, Cybernetics and Informatics*, 2002.
12. Ignacio Silva-Lepe, Revathi Subramanian, Isabelle Rouvellou, Thomas Mikalsen, Judah Diament, and Arun Iyengar. Soalive service catalog: A simplified approach to describing, discovering and composing situational enterprise services. In *Proceedings of ICSOC'08*, pages 422–437. Springer-Verlag, 2008.
13. Natenapa Sriharee and Twittie Senivongse. Matchmaking and ranking of semantic web services using integrated service profile. *Int. J. Metadata Semant. Ontologies*, 1(2):100–118, 2006.
14. Wei-Tek Tsai, Yinong Chen, Raymond Paul, Hai Huang, Xinyu Zhou, and Xiao Wei. Adaptive testing, oracle generation, and test case ranking for web services. In *Proceedings of the 29th Annual International Computer Software and Applications Conference*, pages 101–106, Washington, DC, USA, 2005. IEEE Computer Society.