Multimed Tools Appl
DOI 10.1007/s11042-015-2549-x

Task rescheduling optimization to minimize network resource consumption

Ao Zhou · Shangguang Wang · Ching-Hsien Hsu · Qibo Sun · Fangchun Yang

Received: 7 November 2014 / Revised: 2 February 2015 / Accepted: 5 March 2015
© Springer Science+Business Media New York 2015

Abstract An increasing number of big-data services are being deployed in a cloud computing environment, attracted by the on-demand service, rapid elasticity, and low maintenance costs. As a result, ensuring the quality of service has become an important research problem. Traditionally, task rescheduling is used to ensure a consistent quality of service in the event of failure of a virtual machine. However, the network resource consumption of different rescheduling methods varies. To address this problem, we propose a task rescheduling method that minimizes network resource consumption.The method includes three algorithms. The first obtains a set of good virtual machines from the large quantity of service-providing virtual machines using the skyline operation. A ranking algorithm then fuses the data size and the task emergency to identify significant tasks. Finally, we present an algorithm that automatically determines the optimal insertion point for each task. To verify the effectiveness of the proposed method, we extend the renowned simulator CloudSim and conduct a series of experiments. The results show that our method is more efficient than other methods in terms of network resource consumption.

A. Zhou · S. Wang · Q. Sun · F. Yang
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China

A. Zhou
e-mail: aozhou@bupt.edu.cn

S. Wang
e-mail: sgwang@bupt.edu.cn

Q. Sun
e-mail: qsun@bupt.edu.cn

F. Yang
e-mail: fcyang@bupt.edu.cn

C.-H. Hsu (✉)
Department of Computer Science and Information Engineering, Chung Hua University, Hsinchu 707, Taiwan
e-mail: chh@chu.edu.tw

Published online: 25 March 2015

Keywords Cloud computing · Reliability · Big data analysis · Resource consumption · Task rescheduling

1 Introduction

Cloud computing offers on-demand service, rapid elasticity, and low maintenance costs [2]. On-demand service means that consumers can provision computing, storage,and network resources automatically as needed [5, 19]. Rapid elasticity enables users to acquire and release resources quickly, and low maintenance costs that consumers do not need to maintain their own infrastructure [9, 12]. Attracted by the advantages of cloud computing, an increasing number of big-data industrial companies are migrating their services to the cloud environment [15].

However, cloud services entail a variety of risks [18]. The reliability of the cloud service is greatly influenced by the reliability of the large number of virtual machines (VMs) in the cloud datacenter. The failure of one or more VMs is inevitable in such a complex cloud system [3], and such events are likely to impact all the tasks in the machines waiting queue. Therefore, ensuring the reliability of cloud services has become an urgent research problem.

The inevitability of VM failures means that some fault tolerance method should be adopted to recover the service following a failure event. Fault tolerance is a traditional engineering approach to improving reliability that enables a service to be provided in the event of a failure. The quality demands vary across different services. We consider the following problem. To provide big-data services in a cloud environment, the service-providing VM needs to fetch the data to be processed from the database. There are many big-data processing requests. The huge data from each request consists of many data blocks. To finish each request in time, each data block processing task has a deadline. If a single VM cannot process all the requests, the service is deployed on several VMs. Due to the possible failures of service-providing VM, we shall adopt some failure resilience approach for cloud service. Traditionally, this means rescheduling the tasks to other service-providing VMs. When one VM fails, all tasks in the waiting queue need to be rescheduled to other service-providing VMs. Then, the VM re-fetched the needed data from the central database and restart the task. The process will consume great network resources. However, VM failure events may be caused by software problems. Suppose the failed VM is on host server s. It is likely that s had fetched the necessary data before the VM failed. In this condition, the new VM can fetch the data from s. We can then schedule the task to the nearest service-providing VM to save network resources. However, the rescheduled task may affect the execution of other tasks, and the network resource consumption of different rescheduling method varies.

To overcome the shortcomings, we propose a task rescheduling method (TRM) that combines three algorithms. Because there are multiple service-providing VMs in the cloud datacenter, we first choose a set of ”good” VMs to save time. Therefore, the first algorithm filters the badservice-providing VMs using the skyline operation. A ranking algorithm then fuses the data size with the task importance to identify the most significant tasks. After these two steps, a third algorithm automatically determines the optimal insertion point for each task.

To verify the effectiveness of our method, we extend the renowned cloud simulator CloudSim to obtain FTCloudSim. We implement the proposed approach in FTCloudSim, and compare its performance with that of other methods in terms of the total network

resource consumption. Experimental results show that the proposed method not only ensures cloud service reliability, but also reduces the consumption of network resources.

The rest of this paper is organized as follows. In Section 2, we introduce some related work. In Section 3, further background to the research problem is presented, and the proposed method is described. We introduce the system architecture and additional technical details of our proposed method in Section 4. Section 5 presents the experimental results, and Section 6 concludes the paper.

2 Related work and discussion

There have been plenty of studies on cloud service reliability assurance. Here, we explore the key technologies for ensuring the reliability of cloud services.

A number of researchers have modeled the reliability of cloud services. For example, [13] proposed a scalable method to model the reliability of a large-scale cloud. This paper presented a Markov chain-based method to reduce the analysis time. The cloud system was modeled as a series of coupled interacting Markov chain-based sub models. Furthermore, a Petrinet-based paradigm was presented to solve the Markov chain.

Various types of failure can afflict computing resources, e.g., overflow failure, timeout failure. In [7], these failures were modeled in a cloud computing environment using Markov chains, queueing theory, a Bayesian method, and graph theory. Based on the developed model, the authors also proposed an algorithm to evaluate cloud service reliability.

References [8, 10, 14, 16] adopt fault avoidance techniques to ensure the reliability of cloud services. Fault avoidance techniques try to prevent faults in the system and protect components from failure.

The Cloudval framework [16] was proposed to validate the reliability of cloud infrastructure. This framework tests the virtualized environment by conducting fault injection-based experiments. Fault injection-based testing is a type of black box testing in which users don't need to know the implementation details of the virtualized environment. Various types of fault can be injected, including complex fault models such as maintenance events. The framework is extensible, enabling users to add new fault models.

The effects of temperature on hardware reliability were studied in [8] using a large collection of data. Other methods have aimed to reduce energy consumption and carbon emissions. However, the repeated on-off cycles can increases the probability of host server failure events. The processor life time can also be affected by these on-off cycles. To solve this problem, [10] proposed a tradeoff method to balance energy consumption and reliability cost.

The virtualization of computing resources is implemented by a virtual machine monitor (VMM). As this is software, VMMs face the risk of software aging. To avoid failures caused by software aging, [14] proposed a software rejuvenation method that captures software aging states and enables live migrations. The experimental results showed that cold-VM rejuvenation is sometimes better than warm-VM rejuvenation.

However, the cloud system is very complex, and VM failures are inevitable. Thus, the adoption of a fault tolerance method is necessary. To this end, [10, 14] attempted to ensure cloud service reliability by exploiting redundancy.

There are usually a large number of service requests. Therefore, a service is always deployed across a large number of VMs. In this way, all the service requests can be finished on time. Because some VMs may fail, we need to deploy the service on more VMs than necessary to ensure service reliability. This results in a number of redundant VMs. Indeed,

cloud services with many components will have various levels of redundancy. To attack this problem, [17, 20, 21] proposed a ranking-based method in which all components are ranked according to their invocation structures and invocation frequencies. Based on the ranking results, an optimal algorithm was derived that determines the fault tolerance strategy for different components.

Different from the above method, in which the redundancy of each component is fixed, [11] proposed an unfixed redundancy method. This considers the changing redundancy of each component with time. When a failure event occurs, the method attempts to restore the necessary redundancy. By taking the resource state and service control flow into consideration, the redundancy of service components can be adjusted. This method not only adjusts the components impacted by the failure event, but also reconfigures other components. In this way, the method can reduce the implementation cost of the fault tolerance method.

As we know, when a VM fails, the tasks in the waiting queue need to be rescheduled to other service-providing VMs. However, none of the currently available methods have considered the network resource consumption optimization when rescheduling the affected tasks. Re-fetching the needed data from the central database will consume great network resources. VM failure events may be caused by software problems. We may fetch the data from the host server where the failed VM is placed. However, the network resource consumption of different rescheduling methods is quite different. Our aim is to design a TRM that minimizes network resource consumption.

Unlike the methods discussed in this section, we propose a TRM that can optimize network resource consumption. The details of our method are introduced in the following sections.

3 Preliminaries

To introduce our proposed method, we present a motivating example. Task rescheduling is then formulated as an optimization problem. The notation used throughout the paper is listed in Table 1.

3.1 Motivating example

In cloud computing, all computing resources are virtualized. The computing resources are provided in units of VMs, with one or more VMs hosted on a server. In a cloud environment, services are deployed on many VMs to ensure the large number of tasks finish on time. However, the VMs in the datacenter may fail for many reasons. To ensure reliability, more VMs than needed are provided for the server, and tasks are scheduled to other service providing VMs when a VM fails.

There are many big-data processing requests. The huge data from each request consists of many data blocks. To finish each request in time, each data block processing task has a deadline. As shown in Fig. 1, when a VM fails, all data block processing tasks in the waiting queue need to be rescheduled to a new VM. To finish the request in time, each task has a deadline. If we randomly reschedule tasks to other VMs, they may not be completed on time. If we schedule a task at the end of a waiting queue, that task will not be finished for a long time. If a task is rescheduled at the head of a waiting queue, some other tasks in the queue will not be finished on time. Furthermore, The network resource consumption of different rescheduling methods is quite different. For data-intensive services, the new VM must re-fetch the data to be processed from the central database. As discussed in the

Table 1 Notations

Symbol	Meaning
PM_i	The ith physical machine or host server in the data center, i = 1, 2, ...
vm_F	The failed virtual machine
$PM(vm_F)$	The physical machine on which $PM(vm_F)$ is located
vm_j	A virtual machine, j = 1,2,3 .... vm_j is characterized by a $(PM(vm_i), Q(vm_i))$
T_k	A task, k = 1,2,3.... T_k is characterized by a three-parameter tuple $(R(T_k), t_{deadline}(T_k), VM(T_k))$
$vm(T_k)$	The virtual machine in which T_k is running
$Q(vm_i)$	The task waiting queue of vm_i
L_x	A link in the network, x = 1,2,3
$R(T_k)$	The data size of T_k
$t_{deadline}(T_k)$	The deadline of T_k
$t_{wait}(T_k)$	The queue waiting time
$t_{net}(T_k)$	The queue waiting time
$t_p(T_i, vm_k)$	The time vm_k needed to process T_i
$d(vm_i, vm_j)$	The distance between the two virtual machine
$Pred(Q(vm_i), T_j)$	All tasks before T_j in the queue
$Succ(Q(vm_i), T_j)$	All tasks after T_j in the queue
$Index(Q(vm_i), T_j)$	The position of T_j in the queue

Introduction, the host server s may already have fetched the necessary data before the VM failed, so the new VM can simply fetch the data from s. We can then schedule the task to the nearest service-providing VM to save network resources. We cannot schedule all tasks to the nearest service providing VM. The VM is not strong enough to finish all tasks in time.

To address the above problems, we propose a TRM that minimizes network resource consumption. We consider the problem in which all VM failures are caused by software problems.

3.2 Problem definition

The task rescheduling problem can be formulated as the following optimization problem:

$$min \sum_x \sum_y W_{xy} * R(T_x) \tag{1}$$

$$\begin{gathered} t_{deadline}(T_i) \geq t_{wait}(T_i) + t_{net}(T_i) + t_p(T_i, vm_k), \\ if \quad T_i \subset Q(vm_F) \quad and \quad T_i \quad has \quad been \quad rescheduled \quad to \quad Q(vm_k) \end{gathered} \tag{2}$$

$$\begin{gathered} t_{deadline}(T_j) \geq t_{wait}(T_j) + t_{net}(T_j) + t_p(T_j, vm_k), \\ if \quad T_j \subset Succ(Q(vm_k), T_i) \quad after \quad T_i \quad has \quad been \quad rescheduled \quad to \quad Q(vm_k) \end{gathered} \tag{3}$$

where W_{xy} is 1 if the data of T_x needs to be transferred through network link L_y; otherwise, W_{xy} is 0. The objective function minimizes network resource consumption. The constraint

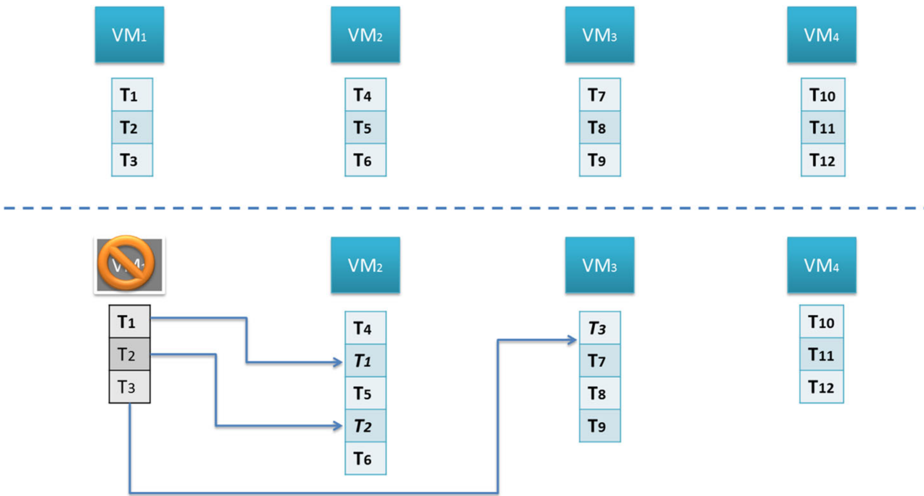

Fig. 1 An example of tasks rescheduling

in (2) ensures that each rescheduled task must be completed before the deadline. The constraint in (3) indicates that the rescheduled task cannot affect the execution of other tasks. After T_i has been rescheduled, all tasks behind T_i must be completed before the deadline.

4 Proposed method

Figure 2 shows the system architecture of our TRM. The main procedures of TRM are as follows:

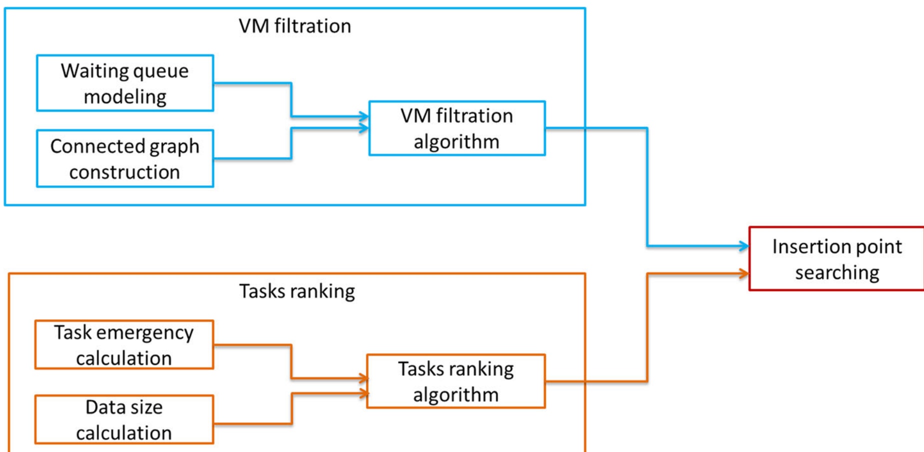

Fig. 2 System architecture of TRM

1. The waiting queue importance is calculated for each service-providing VM. A service-providing VM connection graph is then built based on the datacenter network topology structure.
2. Based on the waiting queue emergency and connection graph, the VMs are filtered by employing the skyline operator.
3. The emergency of each task is calculated. The data size that can be fetched from $PM(vm_F)$ is also obtained.
4. All tasks in the waiting queue of vm_F are ranked.
5. The optimal insertion point for each task in the waiting queue of vm_F is determined.

4.1 Phase 1: virtual machine filtration

The large number of service-providing VMs makes it very time consuming to traverse all VMs. Therefore, we filter out a set of good VMs from the large set of service-providing VMs. We adopt the skyline operation [4] to filter the VMs.

Skyline computation roots in multi-criteria decision making problem, and is commonly used to filter out a set of good points from a large number of data points. Suppose point x and point y are two points in the k-dimensional space. x is denoted by $(x_1, x_2, ...x_k)$, and y is denoted by $(y_1, y_2, ..y_k)$. x dominates y if x is as good or better in all dimensions and better in at least one dimension. Those points which are not dominated by any other point are called skyline. In our method, each point denotes a vm. All VMs in the skyline are called good VM, and other VMs are called bad VMs. In our method, each VM is denoted by a vector. A VM is said to be good if it is not dominated by any other VMs. One VM dominates another if it is as good or better in all dimensions, and better in at least one dimension. In our method, each VM is denoted as follows:

$$vm_i := < e(vm_i), d(vm_F, vm_i) > \tag{4}$$

$$\begin{aligned} e(vm_i) &= \sum_{j=1}^{len(Q(vm_i))} e(T_j) \frac{j}{1+2+3+\ldots+len(Q(vm_i))} \\ &= \sum_{j=1}^{len(Q(vm_i))} e(T_j) \frac{j*2}{len(Q(vm_i))(len(Q(vm_i))+1)} \end{aligned} \tag{5}$$

$$e(T_j) = t_{deadline}(T_i) - t_{current} + t_p(T_i, vm_k) \tag{6}$$

where $e(vm_i)$ denotes the emergency of the waiting queue, and $e(T_j)$ denotes the emergency of each task. If an emergent task is at the head of the queue, we can reschedule a task behind it. If a very emergent task ($e(T_j)$ equal to 0) is at the tail of the queue, we can only reschedule the task at the tail. There is then a large probability that the task cannot be finished on time. Therefore, we calculate the weighted average of the task emergency to obtain the queue importance.

The virtual machine filtration procedure is shown in Algorithm1.

Algorithm 1 Virtual machine filtration algorithm

```
   Input: all virtual machine info vector<e(vm_i),d(vm_F,vm_i)> vms
   Output: all good virtual machines vector<vms> DVms
1  add vms[1] to DVms ;
2  for each element vms[i] in vms do
3  |   for each element DVms[j] in DVms do
4  |   |   if DVms ≺ vms[1] then
5  |   |   |   ;
6  |   |   end
7  |   |   else if vms[i] ≺ DVms[j] then
8  |   |   |   Remove DVms[j] from DVms;
9  |   |   |   Add vms[i] to DVms;
10 |   |   end
11 |   |   else
12 |   |   |   Add vms[i] to DVms;
13 |   |   end
14 |   end
15 end
16 return DVms;
```

Step1: We select a virtual machine from all service-providing VMs, and place it in the skyline set.
Step2: Each service-providing vm_i is compared with all VMs in the skyline set.
Step3: If vm_i is dominated by any VM within the skyline set, it is eliminated and will not be considered in future iterations.
Step4: If vm_i dominates one or more VMs in the skyline set, the dominated VMs are eliminated; that is, these VMs are removed from the skyline set and will not be considered in future iterations. vm_i is inserted into the skyline set.
Step5: If vm_i cannot be compared with all VMs in the skyline set, vm_i is inserted into the skyline set.

At the end of each iteration, we obtain all good VMs (those not dominated by other VMs).

4.2 Phase 2: tasks ranking

The task importance is calculated by (6). When searching for the optimal insertion point, we traverse the VMs, starting with the nearest VM and moving to the furthest VM. VMs that are close to the failed VM will have less network transfer latency. As some tasks are more important than others, we first determine the insertion point for the more important tasks. If the selected VM is close to the failed VM, the data transfer will consume less network resources. Therefore, when two tasks have the same importance, we first search for an insertion point for the task with the larger data size.

Therefore, the tasks are arranged based on their lexicographic sorting order. We sort all tasks according to their importance, with those having the same importance sorted based on their data size.

4.3 Phase 3: insertion point searching

We now describe how the insertion point is determined for each task. The optimal insertion point of a task must satisfy the following constraints: (1) After the task has been inserted into the queue, the rescheduled task can be completed before the deadline. (2) After the task has been inserted into the queue, all the tasks behind the rescheduled task can be completed

before the deadline. When traversing the queues to search for task insertion points, we obtain certain information about the queues, such as the waiting time if we insert the task at the tail of the queue and whether a good insertion point can be determined rapidly in future if queue information is recorded. Algorithms 2 and 3 describe our insertion point search mechanism.

Algorithm 2 Insertion point searching algorithm

```
   Input: all virtual machine info vector<e(vm_i),d(vm_F,vm_i)> vms
   Output: all good virtual machines vector<vms>, DVms
1  for each element T_i in tasks do
2      for each element vm_j in vms do
3          Vector <QI> QInfo = vm_j->QInfo ;
4          if QInfo.size() ≠ 0 then
5              int start = Q(vm_j).size();
6              int end = QInfo.get(QInfo.size());
7              for int k = QInfo.size(); k>0; k– do
8                  QI info = QInfo.get(k);
9                  end = k;
10                 if info− > minDelay < t_p(t_i,vm_j) then
11                     if info− > t_wait < t_deadline(T_i) -t_p(T_i,vm_j) then
12                         bool find = subqueueSearch(vm_j,start,end,T_i);
13                         if find then
14                             goto nextTask;
15                         end
16                         else
17                             goto nextVM ;
18                         end
19                     end
20                     else
21                         goto nextVM ;
22                     end
23                 end
24                 else
25                     bool find = subqueueSearch(vm_j,start,end,T_i);
26                     if find then
27                         goto nextTask;
28                     end
29                 end
30             end
31         end
32         else
33             start = Q(vm_j).size;
34             end = 0;
35             bool find = subqueueSearch(vm_j,Q(vm_j).size,0,T_i);
36             if find then
37                 goto nextTask;
38             end
39             else
40                 goto nextVM ;
41             end
42         end
43         if end != 0 then
44             bool find = subqueueSearch(vm_j,end,0,T_i);
45             if find then
46                 goto nextTask;
47             end
48         end
49     end
50     nextTask
51 end
52 return DVms;
```

Algorithm 3 Subqueue searching algorithm

```
Input: T_i, vm_j, int begin, int end
Output: bool
1  int minDelay = minDelay(Q(vm_j),T_i);
2  int t_wait = t_wait(Q(vm_j),T_i); T_i = Q(vm_j).get(begin);
3  k = begin ;
4  if k ≥ end then
5  |   if t_p(T_i, vm_j) > T_j.getDelay() then
6  |   |   return false;
7  |   end
8  |   if twait > t_deadline(T_i)-t_p(T_i, vm_j) then
9  |   |   k=k-1;
10 |   |   minDelay = min(minDelay, Q(vm).get(k).minDelay);
11 |   |   t_wait = t_wait - t_p(vm_j,T_i);
12 |   end
13 |   else if t_wait < t_deadline(T_i)-t_p(T_i, vm_j) then
14 |   |   insert;
15 |   |   update the information;
16 |   |   return true;
17 |   end
18 end
```

We now traverse the sorted task list. For each task T_i, we traverse the VMs, starting with the closest. For each VM, we examine insertion points from the tail to the head of the queue. If the current waiting time is shorter than the deadline, T_i is inserted into the queue at the current point. Otherwise, if the current waiting time is longer than the time available to the deadline, we move further down the queue to reduce the waiting time. However, this may cause the tasks behind it to miss their deadlines. Therefore, we calculate the waiting time for the following task. If the waiting time is less than that to the deadline, we insert the task and continue the iteration. However, if the waiting time is longer than that to the deadline after the insertion, we break this circle and traverse the next VM.

To reduce the execution time, we record queue information after each task has been inserted, including the waiting time of the inserted task and the most important task behind the inserted task. When traversing a waiting queue, we traverse the inserted tasks from tail to head, and make use of this recorded information to narrow the search.

5 Experiments

To verify the effectiveness of our TRM, we extend CloudSim [6] and conduct a series of experiments. This section describes the experimental setup and presents the experimental results.

5.1 Experimental setup

We construct a 16-port fat-tree datacenter network [1]. Each host server can host up to four VMs. We trigger 100 VM failure events, and extract all tasks with a size of between 5 and 10 min from the DAS2 dataset[1]. We then study the task size distribution, and generate 25000

[1] http://gwa.ewi.tudelft.nl/datasets/gwa-t-1-das2

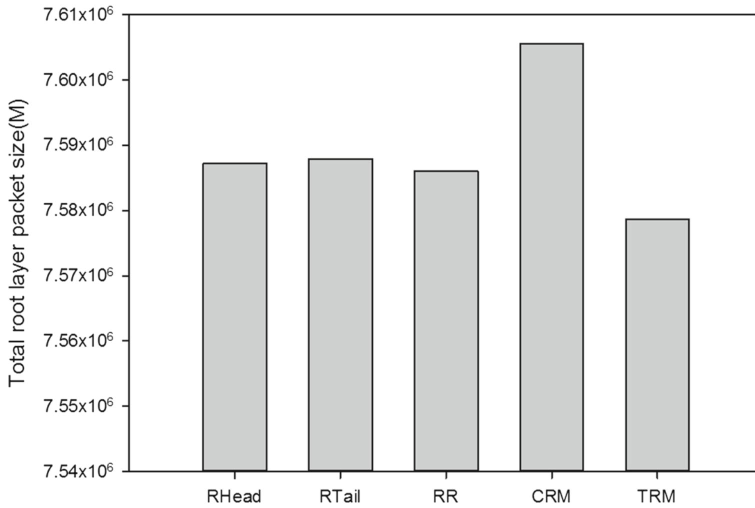

Fig. 3 Total root layer packet size

data block processing tasks with the same distribution. The task size is multiplied by 6to obtain the deadline. The data size of each task is normally distributed between 200 and 400 MB. The number of service-providing VMs for each service follows a normal distribution between 10 and 20. We demonstrate the capabilities of FTCloudSim, which uses our TRM, by comparing it with the following four baseline methods:

1. RHead. A VM is randomly selected for each rescheduled task. The task is inserted at the head of the task waiting queue. The data is fetched from the central database or the host server on which the failed VM located, and the strategy is determined by the network distance.

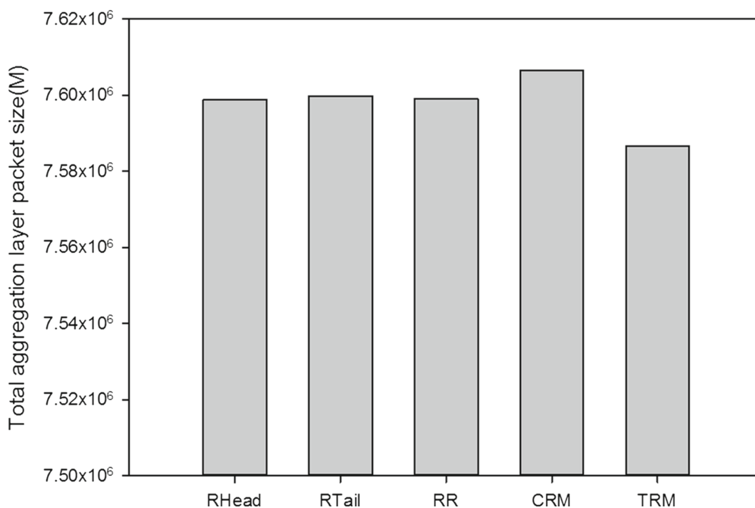

Fig. 4 Total aggregation layer packet size

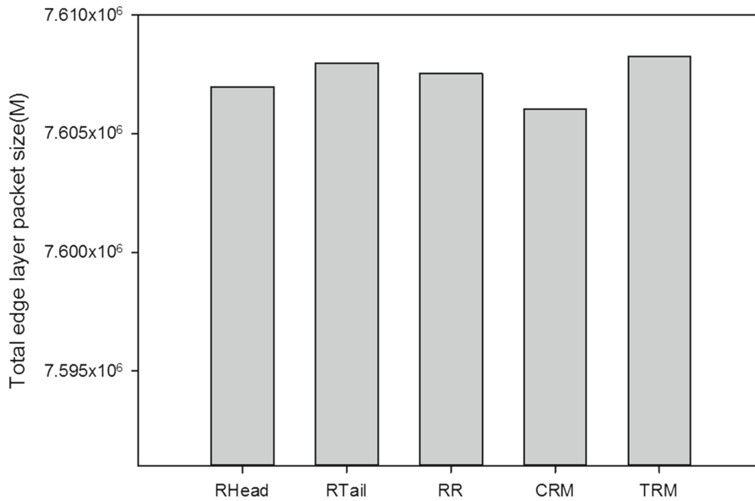

Fig. 5 Total edge layer packet size

2. RTail. A VM is randomly selected for each rescheduled task. The task is inserted at the tail of the task waiting queue. The data is fetched from the central database or the host server on which the failed VM located, and the strategy is determined by the network distance.
3. RR. A VM is randomly selected for each rescheduled task. The task is inserted at random into the task waiting queue. The data is fetched from the central database or the host server on which the failed VM located, and the strategy is determined by the network distance.
4. CRM. The tasks are rescheduled to the nearest VM, but the data is retched from the central database.

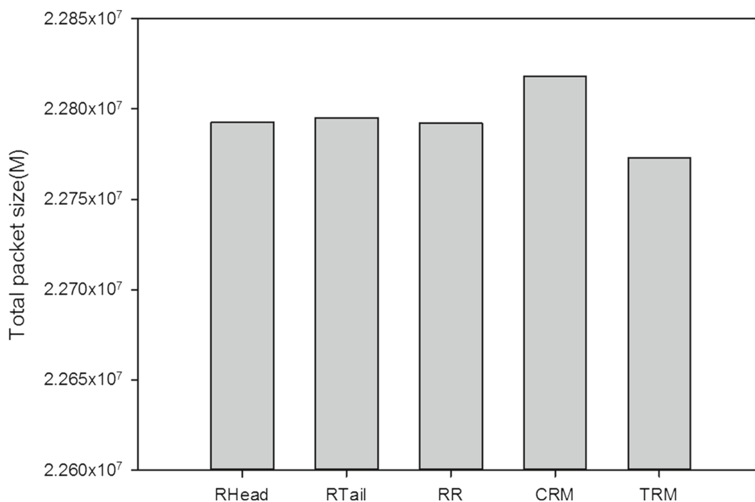

Fig. 6 Total packet size

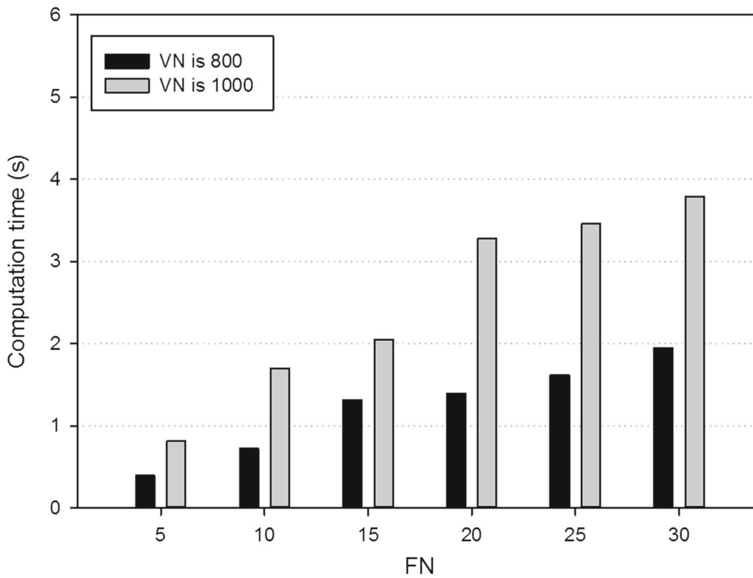

Fig. 7 Computation time

The data is re-fetched from the host of the failed VM if there is one copy. We will evaluate the proposed mechanisms using the following four metrics:

1. Total root layer packet size. The total size of packets that are routed through the root layer.
2. Total aggregation layer packet size. The total size of network packets that are routed through the aggregation layer.
3. Total edge layer packet size. The total size of network packets that are routed through the edge layer.
4. Total packet size. The sum of total root level packet size, total aggregation level packet size and total edge level packet size.

5.2 Performance comparison

Figures 3, 4, 5 and 6 display the network resource consumption performance of all methods on the DAS2 dataset. The results in these figures demonstrate that:

- Compared to other methods, TRM consumes less root level network resources.
- Compared to other methods, TRM consumes less aggregation level network resources.
- CRM consume less edge layer network resource than other methods. That's because when the two data exchanging nodes are in the same pod, the data would be routed through the edge layer twice.
- Compared to other methods, TRM consumes less total network resources.

Because our method considers the network topological structure and task characteristic, tasks are rescheduled to the optimal VMs. Therefore, TRM minimizes network resource consumption.

5.3 Computation time

This section studies the computation time of our method. We investigate how the following two factors affect the computation time of our method: (1) the number of VMs that supply the same service (VN). (2) the number of VMs (of same service) that fail simultaneously (FN). Phase 1 and Phase 2 of our method can execute concurrently. In addition, the rescheduling strategy decision of different services are independent from each other. Therefore, we only count the strategy computation time of a single service. Figure 7 illustrates the computation time of our method. As shown in Fig. 7, the computation time increases as FN increases. In addition, Fig 7 depicts that the computation time is also on an upward trend with an increase in VN. However, comparing to the size of the big-data task, the computation time is very short. The results can validate the effectiveness of our method.

6 Conclusion

In this paper, we have proposed a task rescheduling method to minimize network resource consumption. First, the method filters the ”bad” VMs using the skyline operation. Second, tasks are ranked based on their importance and data size. Finally, a queue searching algorithm determines the optimal location for each task. Experimental results show the advantage of our method.

In future work, we will conduct further experimental analyses on the impact of the number of service-providing nodes, and consider methods of ensuring the reliability of real-time services.

Acknowledgments The work presented in this study is supported by NSFC (61272521); SRFDP (20110005130001); the Fundamental Research Funds for the Central Universities (2014RC1101); Beijing Natural Science Foundation (4132048).

References

1. Al-Fares M, Loukissas A, Vahdat A (2008) A scalable, commodity data center network architecture. ACM SIGCOMM Comput Commun Rev 38(4):63–74
2. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I (2010) A view of cloud computing. Commun ACM 53(4):50–58
3. Bauer E, Adams R (2012) Reliability and availability of cloud computing. John Wiley and Sons
4. Borzsony S, Kossmann D, Stocker K (2001) The skyline operator. In: Proceedings of the 17th International Conference on Data Engineering, IEEE, pp 421–430
5. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Futur Gener Comput Syst 25(6):599–616
6. Calheiros RN, Ranjan R, Beloglazov A, De Rose CA, Buyya R (2011) CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw Pract Experience 41(1):23–50
7. Dai Y-S, Yang B, Dongarra J, Zhang G (2009) Cloud service reliability: Modeling and analysis. In: 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), Citeseer, pp 1–17
8. El-Sayed N, Stefanovici IA, Amvrosiadis G, Hwang AA, Schroeder B (2012) Temperature management in data centers: Why some (might) like it hot. ACM SIGMETRICS Perform Eval Rev 40(1):163–174
9. Fox A, Griffith R, Joseph A, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I (2009) Above the clouds: a Berkeley view of cloud computing. Dept Electrical Eng and Comput Sciences, University of California, Berkeley, Rep UCB/EECS

10. Guenter B Jain N, Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In: Proceedings of the IEEE International Conference on Computer Communications (INFOCOM) IEEE, pp 1332–1340
11. Jung G, Joshi KR, Hiltunen MA (2010) Schlichting, R D, Pu C Performance and availability aware regeneration for cloud based multitier applications. In: Dependable systems and Networks (DSN),IEEE/IFIP International Conference on IEEE, pp 497–506
12. Liu Z, Wang S, Sun Q, Zou H, Yang F (2014) Cost-aware cloud service request scheduling for SaaS providers. Comput J 57(2):291–301
13. Longo F, Ghosh R, Naik VK, Trivedi KS (2011) A scalable availability model for infrastructure as- a-service cloud. In: Proceedings of the 41st annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, pp 335–346
14. Machida F Kim, D S, Trivedi KS (2010) Modeling and analysis of software rejuvenation in a server virtualized system. Software aging,rejuvenation (WoSAR), 2010 IEEE Second International Workshop on IEEE, pp. 1–6
15. Marston S, Li Z, Bandyopadhyay S, Zhang J, Ghalsasi A (2011) Cloud computing? The business perspective. Decis Support Syst 51(1):176–189
16. Pham C, Chen D, Kalbarczyk Z, Iyer RK (2011) Cloudal: A framework for validation of virtualization environment in cloud infrastructure. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN), IEEE, pp 189–196
17. Qiu W, Zheng Z, Wang X, Yang X, Lyu M (2014) Reliability-based design optimization for cloud migration. IEEE Trans Ser Comput 7(2):223–236
18. Schwarzkopf M Murray, D G, Hand S (2012)The seven deadly sins of cloud computing research. In: 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud) pp.1–5
19. Wang S, Liu Z, Sun Q, Zou H, Yang F (2014) Towards an accurate evaluation of quality of cloud service in service-oriented cloud computing. J Intell Manuf 25(2):283–291
20. Zheng Z, Wu X, Zhang Y, Lyu MR, Wang J (2013) QoS ranking prediction for cloud services. Parallel and Distributed Systems. IEEE Trans 24(6):1213–1222
21. Zheng Z, Zhou TC, Lyu MR, King I (2010) FTCloud: A component ranking framework for fault-tolerant cloud applications. In: IEEE 21st International Symposium on Software Reliability Engineering (ISSRE) 2010, IEEE, pp 398–407

Ao Zhou received the M.E. degree in computer science and technology from the Institute of Network Technology, Beijing University of Posts and Telecommunications, in 2012. Currently, she is a Ph.D. candidate at the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. Her research interests include cloud computing, service reliability.

Shangguang Wang is an associate professor at the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. He received his Ph.D. degree in computer science at Beijing University of Posts and Telecommunications of China in 2011. His PhD thesis was awarded as outstanding doctoral dissertation by BUPT in 2012. His research interests include Service Computing, Cloud Services, QoS Management. He is a member of IEEE.

Ching-Hsien Hsu is a professor in department of computer science and information engineering at Chung Hua University, Taiwan. His research includes high performance computing, cloud computing, parallel and distributed systems, ubiquitous/pervasive computing and intelligence. He has been involved in more than 100 conferences and workshops as various chairs and more than 200 conferences/workshops as a program committee member. He is the editor-in-chief of international journal of Grid and High Performance Computing, and serving as editorial board for around 20 international journals.

Qibo Sun received his PhD degree in communication and electronic system from the Beijing University of Posts and Telecommunication in 2002. He is currently an associate professor at the Beijing University of Posts and Telecommunication in China. He is a member of the China computer federation. His current research interests include services computing, internet of things, and network security.

Fangchun Yang received his PhD degree in communication and electronic system from the Beijing University of Posts and Telecommunication in 1990. He is currently a professor at the Beijing University of Posts and Telecommunication, China. He has published 6 books and more than 80 papers. His current research interests include network intelligence, services computing, communications software, soft switching technology, and network security. He is a fellow of the IET.