Cloud Service Reliability Enhancement via Virtual Machine Placement Optimization

Ao Zhou, Shangguang Wang, *Member, IEEE*, Bo Cheng, *Member, IEEE*, Zibin Zheng, *Member, IEEE*, Fangchun Yang, *Senior Member, IEEE*, Rong N. Chang, *Senior Member, IEEE*, Michael R. Lyu, *Fellow, IEEE*, and Rajkumar Buyya, *Fellow, IEEE*

Abstract—With rapid adoption of the cloud computing model, many enterprises have begun deploying cloud-based services. Failures of virtual machines (VMs) in clouds have caused serious quality assurance issues for those services. VM replication is a commonly used technique for enhancing the reliability of cloud services. However, when determining the VM redundancy strategy for a specific service, many state-of-the-art methods ignore the huge network resource consumption issue that could be experienced when the service is in failure recovery mode. This paper proposes a redundant VM placement optimization approach to enhancing the reliability of cloud services. The approach employs three algorithms. The first algorithm selects an appropriate set of VM-hosting servers from a potentially large set of candidate host servers based upon the network topology. The second algorithm determines an optimal strategy to place the primary and backup VMs on the selected host servers with k-fault-tolerance assurance. Lastly, a heuristic is used to address the task-to-VM reassignment optimization problem, which is formulated as finding a maximum weight matching in bipartite graphs. The evaluation results show that the proposed approach outperforms four other representative methods in network resource consumption in the service recovery stage.

Index Terms—Cloud computing, cloud service, reliability, fault-tolerance, datacenter, network resource

1 Introduction

Cloud computing has evolved as an important and popular computing model [1], [2]. Similar to public utility services, computing resources in a cloud computing environment can be provisioned in an on-demand manner [3], [4], and can be purchased via a pay-as-you-go model [5], [6]. This obviates the need to costly over-provision on-premise computing resources to accommodate peak demand [7], [8]. Thus, deploying services into the cloud has become a growing trend [9].

Reliability is an important aspect of Quality of Service (QoS) [10]. With many virtual machines (VMs) running in a cloud datacenter, it is difficult to ensure all the VMs always perform satisfactorily [11]. In reality, many cloud services failed to fulfill their reliability assurance commitment due to VM failures [12]. It is imperative to enhance the reliability of VM-based services in a cloud computing environment [7] [13].

Many solutions have been proposed to address service reliability issues. Fault removal, fault prevention, fault forecasting, and fault tolerance are four basic reliability enhancement techniques [14]. The first three of them attempt to identify and remove faults that occur in the system with the goal of preventing impact-making faults. This goal is unrealistic for a complex computing system like a cloud computing environment in production, in which VM failure is inevitable [15]. Fault tolerance techniques, which try to ensure service continuity when failure occurs, complements those three techniques with a fundamentally different service reliability enhancement approach and with a more practical reliability management goal for cloud services [16], [17].

Many fault tolerance mechanisms have been proposed [18]. Checkpointing [19] is a common fault tolerance mechanism for cloud services. The checkpointing mechanism periodically saves the execution state of a running task (e.g., as a VM image file [20]), and enables the task to be resumed from the latest saved state after failure occurs. However, taking checkpoints periodically and resuming a failed service via checkpoint image(s) are time-consuming. This mechanism may incur too much performance overhead when it is deployed for some small scale tasks or dividable tasks (e.g., a data analytic task that can be divided into a set of small tasks).

Replication [21], e.g., one-to-one and one-to-many standby, is another common fault tolerance mechanism, which exploits redundant deployment of computing resources, e.g., VMs. When the fault tolerance capability of a specific service is provisioned via VM replication, the redundant VMs are classified into two categories: *primary VMs* and *backup VMs*. Notable approaches [22], [23], [24] were developed to reduce the implementation cost by exploiting the degree of redundancy.

- *A. Zhou, S. Wang, B. Cheng, and F. Yang are with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications.*
  E-mail: hellozhouao@gmail.com, {sgwang, chengbo, fcyang}@bupt.edu.cn.
- *Z. Zheng is with School of Data and Computer Science, Sun Yat-Sen University. E-mail: zhzibin@mail.sysu.edu.cn.*
- *R.N. Chang is with IBM Research. E-mail: rong@us.ibm.com.*
- *M.R. Lyu is with Department of Computer Science & Engineering, The Chinese University of Hong Kong. E-mail: lyug@cse.cuhk.edu.hk.*
- *R. Buyya is with Cloud Computing and Distributed Systems (CLOUDS) Lab, Department of Computing and Information Systems, The University of Melbourne, Australia. E-mail: rbuyya@unimelb.edu.au.*

Manuscript received 13 Nov. 2015; revised 5 Jan. 2016; accepted 10 Jan. 2016. Date of publication 20 Jan. 2016; date of current version 8 Dec. 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSC.2016.2519898

k-fault tolerance [25], [26] is a specific type of replication-based fault tolerance mechanism and supports a configuration-based fault-tolerance measurement of a server-based service. A k-fault-tolerant service must be configured with k additional servers such that the minimum server configuration for the service can still be satisfied when k hosting servers fail simultaneously. In a VM-based cloud environment, for example, deploying a specific service on only one server makes the service 0-fault-tolerant (because the service becomes unavailable when the only hosting server fails), regardless the number of redundant VMs that may have been deployed on the same server and the feasibility of restoring the affected service via server reboot or replacement.

When deploying a replication-based fault tolerance mechanism for cloud services, we note that re-assigning an incomplete task from one failed primary VM to a backup VM often requires a huge amount of data to be retrieved and processed once more from the central storage servers. This is a time-consuming and network-resource-consuming process. Moreover, the host server on which a specific failed VM resides may still be running and be able to let the data used by the failed VM accessible from within the VMs running on other servers, e.g., the backup VMs that are in proximity to the failed primary VM. Thus, appropriate VM placement could save considerable amount of time and network resources in failure recovery mode.

Aiming at reducing the lost time and the network resource consumption when the k-fault-tolerance requirement must be satisfied, this paper proposes a novel redundant VM placement approach to enhancing the reliability of cloud services, which is named optimal redundant virtual machine placement (OPVMP). The commercialization nature of network resources in cloud computing prompted us to make OPVMP reduce network resource consumption in addition to enhancing cloud service reliability.

The proposed approach is a three-step process with one algorithm for each of the steps, namely (1) host server selection, (2) optimal VM placement, and (3) recovery strategy decision. The first algorithm selects an appropriate set of VM-hosting servers from a potentially large set of candidate host servers based upon the network topology. The second algorithm determines an optimal strategy to place the primary and backup VMs on the selected host servers. Lastly, a heuristic is used to address the task-to-VM reassignment optimization problem, which is formulated as finding a maximum weight matching in bipartite graphs.

We construct an experimental platform based on our previous research results [27], [28]. Effectiveness of the proposed approach has been evaluated via the platform. The evaluation was done by comparing OPVMP against four other representative redundant VM placement algorithms in terms of four network resource consumption related performance metrics.

All of the five approaches were implemented in FTCloudSim. The evaluation results show that OPVMP outperforms the other four methods in network resource consumption in the service recovery stage.

The remainder of this paper is organized as follows. Section 2 presents a review of related work. In Section 3, the background of the proposed approach is presented. Technical details of the proposed approach are illustrated in Section 4. Experimental evaluation results are reported in Section 5, and the conclusion is presented in Section 6.

2 Related Work

Enhancing the reliability of cloud services is an important aspect of cloud computing and has received considerable attention from the research community. The complex cloud computing environment poses particular challenges to researchers. A variety of service reliability enhancement approaches have been proposed to address related issues.

Checkpointing is a widely used basic fault tolerance mechanism that functions by periodically saving the execution state of a VM as an image file. However, datacenters have limited network resources and may readily become congested when a huge number of checkpoint image files are transferred. Attempting to avoid this problem, Zhang, et al. [29] presented a theoretical delta-checkpoint approach in which the base system only needs to be saved once the first checkpoint completes and subsequent checkpoint images only contain the incrementally modified pages. A theoretical delta-checkpoint approach was implemented by Goiri, et al. in [19]. A further reduction in network resource consumption was developed by Limrungsi, et al. [20], who proposed a peer-to-peer checkpoint approach in which the checkpoint images are stored on the neighboring host servers. If the storage server is located in the same pod as the service-providing server, it is unnecessary to transfer the checkpoint images via core switches.

For some small scale tasks or dividable tasks, for example, Scientific Computing, one huge dataset can be divided into smaller data blocks. Processing each data block would consume much less time than doing that for the huge source dataset. In this case, the cloud supplier may choose other mechanisms in terms of reduction of the checkpointing execution and service resuming overhead.

Replication is another type of reliability assurance mechanism. Replication is based on the exploitation of redundancy. One-to-one and one-to-many standbys are two well known mechanisms. Xu , et al. in [21] tried to map each primary VM to a backup VM. A primary VM and its mapping backup node form a survivable group. A task can be completed in time if at least one VM in the survivable group works well. The work takes the bandwidth reservation into consideration when solving the mapping problem. In this regard, an optimal algorithm that maps a survivable group to the physical data center was proposed [21].

How to reduce the cost of replication is a problem that has been addressed by several proposals [22], [23], [30]. All of these proposals aim to reduce the degree of redundancy. Another effort [30] was based on the fact that different modules have different redundancy requirements and that modules with a higher invocation frequency are more significant than other modules. Thus, the work attempted to rank all the modules of a system based on their significance value. The same problem was approached differently [24] by adjusting the redundancy of the same module under different execution conditions.

k-fault tolerance [25], [26] is another approach aiming at reducing the cost of implementing redundancy. k-fault tolerance ensure that the simultaneous failure of any k computing nodes would not make the service unavailable.

TABLE 1
Notations

Symbol	Meaning
PM_i	The physical machine or host server in the data center, $i = 1, 2, \ldots$
VM_j	The virtual machines in the data center, $j = 1, 2, \ldots$
pod_x	The pods in the data center, $x = 1, 2, \ldots$
T_y	The task submitted by users, $y = 1, 2, \ldots$
$subnet_l$	The subnet in the data center, $l = 1, 2, \ldots$
max_subnet	The number of subnets which contain available host servers
S	A service
$VM_P(S)$	Return the primary VMs of S
$VM_B(S)$	Return the backup VMs of S
$VM_F(S)$	Return current failed VMs of S
$PM(S)$	Return all the servers on which service providing VMs of S locate
$Succ$	The next element after current element
$size$	Return the element number of a list
$DSize$	Return the size of the data stored in a server
a	The number of available PM in a subnet
m	The number of primary VM of S
k	The number of backup VM of S
min	Return the min value of the inputs
$length$	The linkage length
$copy$	Copy all data from a vector to another vector

There is a large quantity of tasks. To complete all tasks in time, a service would be deployed in several VMs. The tasks are scheduled to the VMs according to an appropriate scheduling strategy. In a cloud computing environment, the computing resources of each server are virtualized to several VMs. Both hardware and software problems can result in VM failures. When a host server crashes, all the VMs it is hosting will no longer operate. The more the number of VMs providing the same service is placed on the same server, the more serious the damage is when a host server fails. Taking this into consideration, Machida, et al. in [25] proposed a redundant VM placement approach to ensuring k-fault tolerance.

However, when restarting a task from one backup VM, we need to re-fetch the data to be processed from the central database. The process is time-consuming and network resource-consuming for cloud services. We note that the host server on which the failed VM runs may store a copy of data for the task. If the backup VM can fetch the data it needs directly from that host server despite of the unexpected primary VM failure, a lot of data processing time and network resources can be saved if the primary VMs and the backup VMs are properly allocated.

One major difference between our proposed cloud service reliability enhancement approach and related work is that we use a redundant VM placement method and optimize network resource consumption through appropriate placement of the required VMs.

3 Preliminaries

This section provides the background and motivation for our work as well as explains how we formulate the redundant VM placement problem as an optimization problem. The notations listed in Table 1 will be used throughout the rest of the paper.

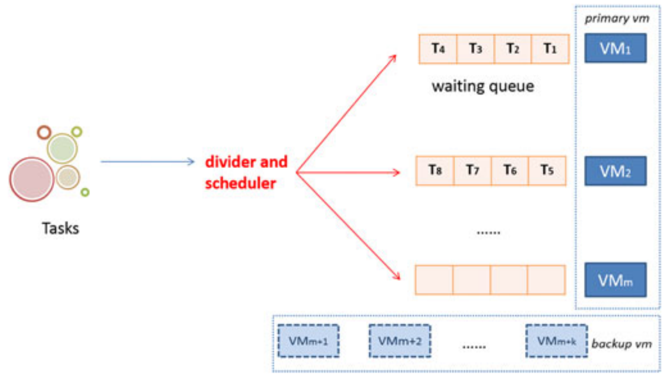

Fig. 1. Task processing model.

3.1 Background

In the cloud computing environment, a statistically rare failure event may be a common occurrence due to scale [31]. Cloud service reliability enhancement is becoming an important research challenge.

Fig. 1 shows the task processing model we use. The cloud service is employed in several VMs because, considering the huge amount of service requests, the computing power of a single VM would be insufficient. Upon receiving the service requests, the divider partitions the large scale task into smaller sub-tasks. Processing each sub-task would not consume too much time. Based on the scheduling algorithm, each task to one of the service-providing VMs. Each VM has a task waiting queue. Since a VM may fail due to a software or hardware fault, an assigned task may not be completed as scheduled, and may in turn delay the entire service request operation. k-fault tolerance [25], [26] can be chosen by the cloud service provider to reduce lost time and to ensure service reliability. Besides the m number of primary VMs, there are k backup VMs for each service. k-fault tolerance serves to ensure that the task processing service will not be down in the event of the simultaneous failures of any k computing VMs or servers. All the primary and backup VMs are placed on different host servers. Otherwise, when a host server crashes, all the VMs it is hosting will no longer operate. Failures of one of the primary VMs result in it being mapped to a backup VM, and the tasks in its waiting queue are reassigned to the backup VM.

3.2 Motivation

Suppose there are m virtual machines that provide service S. All tasks of S are assigned to the m virtual machines based on appropriate strategy. When k-fault tolerance replication is adopted to ensure the service reliability, there are k backup virtual machines for S. To avoid both hardware and software problems that lead to VM failures, the primary and backup virtual machines are distributed to different servers. When a primary VM fails, it is mapped to a backup VM, and the tasks in its waiting queue are re-assigned to the backup VM. The backup VM need to re-fetch the data from the database. This scheme has two shortcomings. First, the storage servers are continually required to process a huge number of data read and write requests, which is sometimes time consuming [20]. Second, transferring large amounts of data consumes considerable network resources. When a VM failure event is caused by hardware problems,

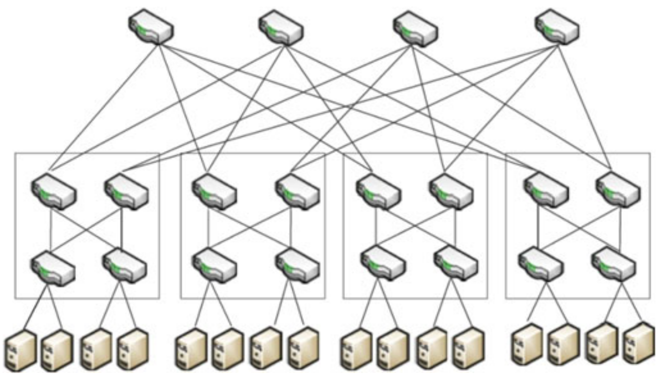
Fig. 2. Fat-tree data center network with four ports.

the mapped backup virtual machine re-fetches the needed data from the database. When a VM failure event is caused by software problems and the server which hosts the failed VM has a copy of data, the backup VM can fetch the data from the server. More network resources can be saved if the failed primary VM and the backup VM are in proximity to each other. Therefore, appropriate VM placement could save considerable amount of time and network resources in the failure recovery mode. To save more network resources, we place the primary and backup virtual machines by considering the network topology of the datacenter.

Datacenter networks always adopt a tree-like structure [32] of which the fat-tree network structure is a commonly used datacenter network architecture [33]. A fat-tree network typically consists of trees with three levels of switches [34] as illustrated in Fig. 2, in which a fat-tree datacenter network with four ports is depicted. The switches in the top, middle, and bottom layers are referred to as root, aggregation, and edge switches, respectively. The host servers physically connect to the network via the edge switches. All the host servers sharing the same edge switch with each other are addressed in the same subnet. All host servers sharing the same aggregation switches are addressed in the same pod. Upper layer switches transfer data from more host servers, and are therefore more likely to become congested than the lower layer switches [35]. Therefore, reducing the network resource consumption of the upper layer links becomes an important problem that has to be solved [20]. The host servers must retrieve data from the storage servers, all of which are connected by the storage area network (SAN). In turn, each one of these storage resources is segmented into a number of virtual disks [36]. In a centralized storage scheme where the SAN switches connect to the root layer switches [20], re-fetching the data from the storage server may consume too much upper layer resources. As described before, the server hosting the failed VM may store a copy of the data for the affected tasks. If the VM failure was caused by a software problem, the data may be retrieved from the host server on which the failed VM is placed. If the primary and the backup VMs are in the same subnet, the transfer only consumes the edge-level network resource. However, if the primary and the backup VMs are in the same pod, the transfer will consume both the edge-level and the aggregation-level network resources.

Thus, appropriate VM placement would save time and network resources. As indicated above, the best solution would be to place all the primary and backup VMs on host servers in the same subnet. However, this may not be possible if some of the host servers in the datacenter have already been allocated to other tasks and have insufficient free computing resources. Alternatively, a subnet may not even contain a sufficient number of available host servers. Therefore, it may be necessary to place the $(m+k)$ VMs in different subnets. The problem becomes complex now because different VM placement strategies could result in different network resource consumption. Suppose a service needs (2+2) VMs for two-fault tolerance and there are two available subnets in the same pod, each with two available host servers. There are two placement strategies: (1) the two primary VMs are placed on host servers in subnet 1, whereas the backup VMs on host servers in subnet 2; (2) one primary and one backup VM are placed on host servers in subnet 1, whereas the remaining two VMs on host servers in subnet 2. In strategy 2, the data transfer only consumes edge level network resources in the recovery stage; therefore, strategy 1 will consume more network resources than strategy 2. When k is equal to m, the problem can be solved by minimizing the difference of the number of primary and backup VMs in each subnet. However, the problem becomes more complex when k is smaller than m.

To address this problem, our approach aims to determine an optimal placement strategy to enhance the service reliability and minimize the network resource consumption.

3.3 Problem Definition

The redundant VM placement problem can be formulated as the following optimization problem:

$$min UP(S) and UD(S) \tag{1}$$

subject to:

$$UP(S) = \sum_i \sum_j (DSize(pkt_i)) * w_{ij} \tag{2}$$

$$UD(S) = \sum_i \sum_j delay_{ij} \tag{3}$$

$$w_{ij} \in \{0, 1\} \tag{4}$$

$$\sum_i x_i = m \tag{5}$$

$$x_i \in \{0, 1\} \tag{6}$$

$$\sum_i y_i = k \tag{7}$$

$$y_i \in \{0, 1\} \tag{8}$$

$$\sum_i z_i = m + k \tag{9}$$

$$z_i \in \{0, 1\}, \tag{10}$$

where $UP(S)$ denotes the total network consumption. The value of w_{ij} is 1 if pkt_i transfers through $link_j$. When the replication strategy is adopted, the downtime is mainly affected by the data transfer delay. $UD(S)$ denotes the total data transfer delay. $delay_{ij}$ denotes the transfer delay of pkt_i transferring through $link_j$. The constraints in (5) and (6) ensure there are m primary VMs for the service. The constraints in (7) and (8) ensure there are k backup VMs for the service. The constraint in (9) and (10) ensures the (m+k) VM are all placed on different host servers, therefore, the placement strategy ensures k-fault tolerance irrespective of whether the failure of the VMs is caused by a software fault or host server fault.

Suppose the number of subnets which contain available host servers is max_sub, then the number of available host servers in each subnet is stored by using the following vector:

$$A = [a_1, a_2, \ldots, a_{max\_subnet}]. \tag{11}$$

The solution to the problem can be defined by the following two vectors:

$$M = [m_1, m_2, \ldots, m_{max\_subnet}], \tag{12}$$

$$K = [k_1, k_2, \ldots, k_{max\_subnet}], \tag{13}$$

where (12) denotes the number of primary VMs in each subnet, and (13) denotes the number of backup VMs in each subnet. Therefore, it is necessary to obtain an optimal solution by considering all the solutions of the following indeterminate equations:

$$\begin{cases} m_1 + m_2 + \ldots + m_{max\_subnet} = m \\ k_1 + k_2 + \ldots + k_{max\_subnet} = k \\ m_i + k_i \le a_i, \quad i = 1, 2, 3 \ldots, max\_subnet \\ m_j \ge 0, \quad i = 1, 2, 3 \ldots, max\_subnet \\ k_i \ge 0, \quad i = 1, 2, 3 \ldots, max\_subnet. \end{cases} \tag{14}$$

There is a huge number of pods, subnets and host servers in a cloud datacenter. Iterating over all the placement strategies would be intractable; therefore, the problem is solved by adopting a heuristic optimal algorithm, as discussed in the next section.

4 Proposed Approach

The formulated problem essentially involves finding (k+m) host servers followed by placing (k+m) VMs on those host servers. Since there are a huge number of host servers in a cloud datacenter, the possible number of solutions is exponentially large. It is consequently necessary to identify a subset of good host servers from which to obtain the best solutions. The procedure that was used to select (m+k) good host servers is provided in Section 4.1 and the algorithm used for placing (m+k) VMs on those host servers is presented in Section 4.2. Given the information about the failed and the backup VMs, a recovery strategy decision algorithm calculates the optimal matching strategy. The proposed recovery strategy decision algorithm will be discussed in Section 4.3.

4.1 Phase 1: Host Servers Selection

As explained in Section 3.2, the further the two host servers are located from one another, the greater the delay becomes. In addition, the data transfer traffic would consume more network resources. To avoid this situation, it is desireable to place all VMs in a subnet that contains (m+k) available host servers. Suppose there are two subnets, one of which has (m+k+20) available host servers, and the other has (m+k+1) host servers. In cases such as these, our approach would choose the second option and leave the first option to another service that may require more host servers. In other words, we would follow the "just enough rule". The "just enough rule" means that we select the pod or subnet with just enough resource, and leave the residual capacities for future use. If none of the subnets have a sufficient number of available host servers, the VMs must be distributed to several subnets or even several pods. In this case, those pods with a greater number of available host servers will be considered first to avoid traffic between pods in the recovery stage. We also consider selecting the subnets based on the above rule. The host server selection procedure is shown in Algorithm 1 as explained in the following steps:

Step 1: Sort all subnet based on available host servers. A host server is "available" when the server has sufficient computing resources to host the virtual machine. Search for a subnet $subnet$ that satisfies the "just enough rule". Select (m+k) servers from $subnet$ and assign them to servers, and return. (Lines 1 to 4 and Lines 49 to 53)

Step 2: Add all pods in the datacenter to a list, and sort the list according to the available host servers. Assign the head of the list to variable $HPod$. If the number of available host server of $HPod$ is larger than (m+k), goto Step 6. Else, goto Step 3. (Lines 5 to 7)

Step 3: Add all host servers in the pod to $servers$. Iterate the pod list and collect host servers until sum of $size$($servers$) and the available host servers in current pod is larger than (m+k). If size($servers$) is equal to (m+k), return. (Lines 15 to 26)

Step 4: Sort all subnets in current pod according to the available host servers. Assign the head of the list to variable $HSubnet$. If the number of available host server of $HSubnet$ is larger than (m+k), goto Step 5. Else, goto Step 8. (Lines 27 to 33)

Step 5: Iterate the subnet list. Search for a subnet $HSubnet$ that satisfies the "just enough rule". Select (m+k) host server from the subnet and return. (Lines 49 to 52)

Step 6: Continue to iterate the pod list and search for a pod pod that satisfies the "just enough rule". If the number of available host servers is equal to (m+k-$size$($servers$)), add all host server in the pod to servers, and return. (Lines 8 to 12)

Step 7: Add all subnets in the pod to a list, and sort the list according to the available host servers. Assign the head subnet of the list to variable $HSubnet$. If the number of available server is larger than (m+k-size($servers$)), goto Step 9. Else, goto Step 8. (Lines 28 to 33)

Step 8: Add all host servers in $HSubnet$ to servers. Iterate the subnet list and collect host servers until sum of $size$($servers$) and the available host servers in current subnet is larger than (m+k). If $size$($servers$) is equal to (m+k), return. (Lines 35 to 44)

Step 9: Continue to iterate the subnet list. Search for a subnet $subnet$ that satisfies the "just enough rule". Select (m+k) from subnet and assign them to servers, and return. (Lines 49 to 52)

The iterations enable all the "good" host servers to be obtained. Section 4.2 describes the procedure that was used to place the (m+k) VMs on the (m+k) host servers.The time complexity of Algorithm 1 is O(nlogn), and n is the number of subnets in the datacenter.

Algorithm 1. Host Server Selection

```
Input: the number of needed primary VMs $m$,the number of
       needed backup VMs $k$
Output: list of interesting host servers $servers$
1  sort all subnets by the number of available host servers and
   $subnet = subnets->head$;
2  if ($subnet->freeServerSize$)≥ ($m$+$k$) then
3      goto final2;
4  end
5  sort $pods$ by the number of available host servers;
6  $pod = pods.head$;
7  if ($pod.freeServerSize$)≥ ($m$+$k$) then
8      $next$ = Succ($pod$);
9      while ($next.freeServerSize$)≥ ($m$+$k$) do
10         $pod = next$ and $next$ =Succ($pod$);
11     end
12     add all subnets in $pod$ to $subnets$ and goto final1;
13 end
14 else
15     add all available host servers in $pod$ to $servers$;
16     while size($servers$) < ($m$+$k$) do
17         $pod$ = Succ($pod$);
18         if size($pod.freeServerSize$)+size($servers$) > ($m$+$k$) then
19             add all subnets in $pod$ to $subnets$;
20             goto final1;
21         end
22         else
23             add all available host servers in $pod$ to $servers$;
24         end
25     end
26 end
27 final1:
28 sort $subnets$ by the number of available host servers;
29 $subnet = subnets.head$;
30 final2:
31 if ($subnet.freeServerSize$)+ size($servers$)≥ ($m$+$k$) then
32     goto final 3;
33 end
34 else
35     add all available host servers in $subnet$ to $servers$;
36     while size($servers$) < ($m$+$k$) do
37         $subnet$ = Succ($subnet$);
38         if ($subnet.freeServerSize$)+size($servers$) ≥ ($m$+$k$) then
39             goto final3;
40         end
41         else
42             add all available servers in $subnet$ to $servers$;
43             return $servers$;
44         end
45     end
46 end
47 final3:
48 $next$ = Succ($subnet$);
49 while ($next.freeServerSize$)+ size($servers$) ≥ ($m$+$k$) do
50     $subnet = next$ and $next$ =Succ($subnet$);
51 end
52 select $(m + k)$- size($servers$) available servers from $subnet$,
   and assigned them to $servers$;
53 return $servers$;
```

4.2 Phase 2: Virtual Machine Placement

Placing (m+k) VMs on the (m+k) host servers requires the number of backup and primary VMs in each subnet to be determined.

A heuristic algorithm is used to solve this problem. Two heuristic conditions are adopted to narrow the searching space. If there are even number of available servers in a subnet, the number of backup vm in the subnet should be less or equal to the number of primary vm in the subnet. If there are odd number of available servers in a subnet, the number of backup vm in the subnet should be less or equal to one plus the number of primary vm in the subnet. Suppose there is a subnet that contains fewer primary VMs than backup VMs. As the total number of primary VMs is larger than or equal to the number of backup VMs, there is at least one subnet in which the number of backup VMs is smaller than the number of primary VMs. Now, a backup VM in the first subnet and a primary VM in the second subnet exchange position with each other. Compared to the first strategy, one more failed VM in the second subnet does not require to be mapped to a backup VM in different subnets when k number of VMs fail at the same time. The new placement strategy will consume less aggregation layer network resource. The second heuristic condition is that the subnet that contains more available host servers should be allocated more backup VMs. If a subnet that contains more available host servers, the difference between the number of primary VMs and backup VMs of it should be smaller. When the difference between the number of primary VMs and backup VMs is larger, there is a larger chance that the data transfer would consume more network resources. The proof of the first heuristic condition can be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSC.2016.2519898. The proof of the second heuristic condition can be found in Appendix B, available in the online supplemental material.

Our algorithm is shown in Algorithm 2 and 3, which are recursive in nature. In each recursion, the algorithm determines the number of backup VMs that are placed in the current subnet, and the rest backup VMs are placed in the following subnets. When the number of rest backup VM equals 0 or the last subnet has reached, the resource consumption of the current placement strategy is computed and compared with the current optimal one. If the resource consumption is smaller than that of the current optimal strategy, the current strategy is considered optimal. When the current placement cannot satisfy the two heuristic conditions, the algorithm terminates current recursion and backtrace.

When a backup VM fails, a new backup VM is searched around the old one.

4.3 Phase 3: Recovery Strategy Decision

When one or more VMs fail, a recovery strategy has to be decided upon, and each failed VM has to be mapped to a backup VM. All tasks in the waiting queue of the failed VM are rescheduled to its mapping backup VM, and the data to be processed have to be retrieved again to the backup VM. If the VM fails because of a software fault, the particular data block may be obtained from the host server on which the failed VM resides. Given the information on the VM failure caused by a software fault and the backup VMs, the recovery strategy decision algorithm matches the failed VMs and the backup VMs. Then each failed VM has to be mapped to a backup VM. The recovery strategy should minimize the total network resource consumption. In this case it is possible to formulate the recovery strategy decision problem as a minimum weight matching in bipartite graphs [37], [38].

Algorithm 2. Virtual Machine Placement on Specific Servers

```
Input: interesting host servers $servers$, the number of backup
       servers $k$
Output: int $strategy[]$
1 Obtain all the "good" subnets that at least have one "good"
  host server;
2 Store the "good" host server number of each interesting sub-
  net to vector $subnets[]$;
3 Sort $subnets[]$ by the number of "good" host server desc;
4 int $strategy$[size($subnets$)];
5 int $optimal$[size($subnets$)];
6 int $mincost = \infty$;
7 optimal placement strategy searching($subnets$, $strategy$,
  $optimal$, $k$, $mincost$, 1);
8 return $strategy$;
```

Given a complete bipartite graph G=(V,E) with bipartition(V_F,V_B), where V is the set of all failed VMs and backup VMs, V_F is the set of all failed VMs, V_B is the set of all backup VMs, and E is the set of shortest paths connecting nodes for each pair of VMs from different partitions. A matching set M is a subset of E. Suppose w denotes the weight function, and then it is necessary to find a matching of minimum weights where the weight of matching M is given by:

$$w(M) = \sum_{e \in M} (w(e)). \tag{15}$$

In other words, the recovery problem can be formulated as the following:

$$min \sum_{(V_F, V_B)} (w(v_f, v_b) * x(v_f, v_b)) \tag{16}$$

subject to:

$$\sum_{(v_f)} (x(v_f, v_b) = 1), \forall v_f \subset V_F \tag{17}$$

$$x(v_f, v_b) \in 0, 1, \forall v_f \subset V_F, \forall v_b \subset V_B \tag{18}$$

$$w(v_f, v_b) = DSize(v_f) * length(e(v_f, v_b)), \tag{19}$$

where (17) ensures each failed VM is matched to a backup VM. In (18), $x(v_f,v_b)$ = 1 if the edge (v_f,v_b) belongs to the matching; otherwise, $x(v_f,v_b)$ = 0. $DSize$ in (19) returns the data size that can be retrieved from the host server on which v_f is placed. Therefore, $w(v_f,v_b)$ denotes the transfer cost.

Algorithm 3. Optimal Placement Strategy Searching

```
Input: $subnets$, $strategy$, $optimal$, the number of un-placed
       backup VMs $rest$, $mincost$, nested level $i$
Output: The placement strategy
1  if $rest == 0$ then
2      $strategy$[i] = $rest$;
3      Compute cost of current strategy;
4      if $currentcost \leq mincost$ then
5          $minCost$=$currentcost$;
6          copy($strategy$[], $optimal$[]);
7      end
8      return;
9  end
10 if $i$ == $size(subnets)$ then
11     if $rest > min(strategy[i-1], ceil(subnet[i]/2))$ then
12         return;
13     end
14     $strategy$[i] = $rest$;
15     Compute cost of current strategy;
16     if $currentcost \leq mincost$ then
17         $mincost$=$currentcost$;
18         copy($strategy$[], $optimal$[]);
19     end
20     return;
21 end
22 $n$ = min($strategy$[i-1], ceil($subnets$[$i$]/2), $rest$);
23 while $n \geq 0$ do
24     $strategy$[i]=$n$;
25     optimal placement strategy searching($subnets$,$strategy$,
       $optimal$,$k$-$n$,$cost$);
26     $n$–;
27 end
28 return;
```

As explained before, information that is exchanged between host servers in the same subnet only utilizes an edge switch; however, when two hosts are in the same pod, all communicated traffic is routed through both the edge and the aggregation switches. Therefore, the transfer will consume more network resource and the delay becomes greater. For each backup VM, we try to find a corresponding failed VM in the same subnet for it if there is one. Otherwise, VMs in different subnets would have to be mapped.

Algorithm 4 details our recovery strategy decision algorithm:

Step 1: For each backup VM, all failed VMs in the same subnet that have not been matched are sorted according to their data size. The backup VM is mapped to the failed VM with the largest data size.

Step 2: If there still remain failed VMs that have not been matched to a backup VM, all unmatched failed VMs in the same pod are sorted according to data size. The backup VM is mapped to the failed VM with the largest data size in the pod.

Step 3: If there still remain failed VMs that have not been matched to a backup VM, the backup VMs are randomly matched to the failed VMs. In addition, the data are re-fetched from the storage server.

If we iterate all subnets concurrently, the time complexity of Algorithm 4 is O(nlogn). n is the number of failed VM. We note that the proposed network-topology-aware redundant VM placement approach aims at enhancing the reliability of server-based cloud services whose fault-tolerance level can be measured and assured in terms of the k-fault-tolerance metric. In practice, rebooting or replacing a specific failed VM on the same hosting server could be attempted in a controlled manner (as a precondition for executing the aforementioned recovery strategy) as a VM-specific optimization of the proposed approach, though the VM failure handling scheme is not helpful to the k-fault-tolerance measurement. The scheme needs be done in a controlled manner because service-specific VM management policy may need be followed, particularly when the root cause is unknown (which may lead to continual failure recovery attempts caused by partial server failures).

Algorithm 4. Recovery Strategy Decision

```
Input: Set of all failed host servers $VM_F$, set of all backups
       $VM_B$, $w(vm_f,vm_b)$ for each $vm_f \in VM_F$ and $vm_b \in VM_B$
Output: Map $maps$ between each failed host server and its
        backup
1  add all subnets that contains at least one backup VM to
   $subnets$ ;
2  for each subnet $subnet_i$ in $subnets$ do
3    add $VM_F \cap VM(subnet_i)$ to list $srcVM$;
4    add $VM_B \cap VM(subnet_i)$ to list $dstVM$;
5    sort $srcVM_f$ by data size;
6    while $srcVM$ is not $\emptyset$ and $dstVM$ is not $\emptyset$ do
7      $key$ = $srcVM$-> $head$;
8      $value$ = $dstVM$-> $head$;
9      add < $key$ ,$value$ > to maps;
10     remove $key$ from $PM_F$ and $srcVM$ ;
11     remove $value$ from $PM_B$ and $dstVM$ ;
12   end
13 end
14 if $VM_F$ is not $\emptyset$ then
15   for each pod $pod_i$ in pods do
16     clear $srcVM$ and add $VM_F \cap VM(pod_i)$ to list $srcVM$;
17     clear $dstVM$ add $VM_B \cap VM(pod_i)$ to list $dstVM$;
18     while $srcVM$ is not $\emptyset$ and $dstVM$ is not $\emptyset$ do
19       $key$ = $srcVM$-> $head$;
20       $value$ = $dstVM$-> $head$;
21       add < $key$ ,$value$ > to maps;
22       remove $key$ from $PM_F$ and $srcVM$ ;
23       remove $value$ from $PM_B$ and $dstVM$ ;
24     end
25   end
26 end
27 if $VM_F$ is not $\emptyset$ then
28   while$VM_F$ is not $\emptyset$ do
29     $key$ = $VM_F$ -> $head$;
30     $value$ = $VM_B$ -> $head$;
31     add < $key$ ,$value$ > to maps;
32     remove $key$ from $PM_F$;
33     remove $value$ from $PM_B$;
34   end
35 end
36 return $maps$;
```

5 Experimental Evaluations

In the following sections, the experimental setting is first outlined. Then OPVMP is compared with other four representative approaches in terms of the total network resource consumption and other performance metrics. Finally, the parameters of our approach are studied.

5.1 Experimental Setup

We construct an experimental platform based on our previous research results [28], [39]. In our experiment, a 32-port fat-tree data center network is constructed. The capacity of the root-layer link and aggregation-layer link is set as 10 Gbps, and the capacity of the edge-layer link is set as 1 Gbps [21]. There are 16 host servers in each subnet. Each of these host servers can host four VMs at most. The performance of our method (OPVMP) was studied by comparing it with four other existing representative methods:

- *RSVMP*. Data are re-fetched from the central storage server in the recovery stage. This approach does not take the network topology into consideration. After having selected (m+k) host servers, all primary VMs and backup VMs are randomly placed on the selected host servers in the data center [25].
- *RLVMP*. Data are re-fetched from the central storage server or the host server on which the failed VM resides. The strategy is determined by the network distance and whether a data copy exists. After having selected (m+k) host servers, all primary VMs and backup VMs are randomly placed on the selected host servers.
- *PLVMP*. Data are re-fetched from the central storage server or the host server on which the failed VM resides. The strategy is determined by the network distance and whether a data copy exists. After having selected (m+k) host servers, the backup VMs are uniformly distributed across the pods.
- *SLVMP*. Data are re-fetched from the central storage server or the host server on which the failed VM resides. The strategy is determined by the network distance and whether a data copy exists. After having selected (m+k) host servers, the backup VMs are uniformly distributed across the subnets.

All the methods were evaluated using the following performance metrics:

- *TDelay:* Total data transfer delay, $TDelay$, can be calculated as follows:

$$TDelay = \sum_i delay(pkt_i), \tag{20}$$

  where pkt_i denotes a network packet.
- *PRoot:* The total size of network packet that has been transferred by the root layer switches. $PRoot$ can be calculated as follows:

$$PRoot = \sum_i w_r \times size(pkt_i), \tag{21}$$

  where w_r denotes the frequency with which packet pkt_i has been transferred by the root switches.
- *PAgg:* The total size of network packet that has been transferred by the aggregation layer switches. $PAgg$ can be calculated as follows:

$$PAgg = \sum_i w_a \times size(pkt_i), \tag{22}$$

  where w_a denotes the frequency with which packet pkt_i has been transferred by the aggregation switches.

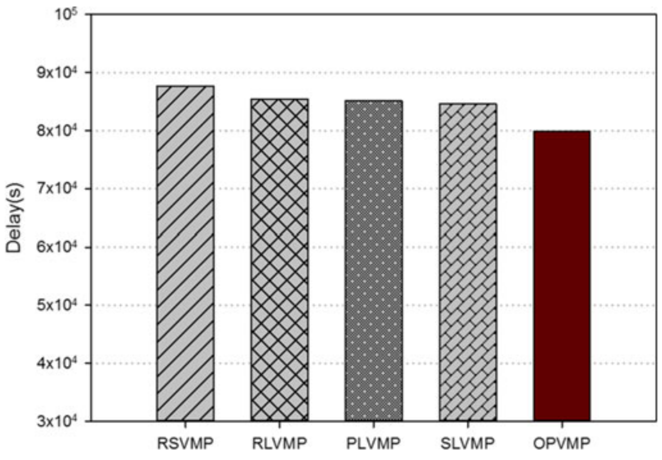

Fig. 3. The performance of total data transfer delay (s).

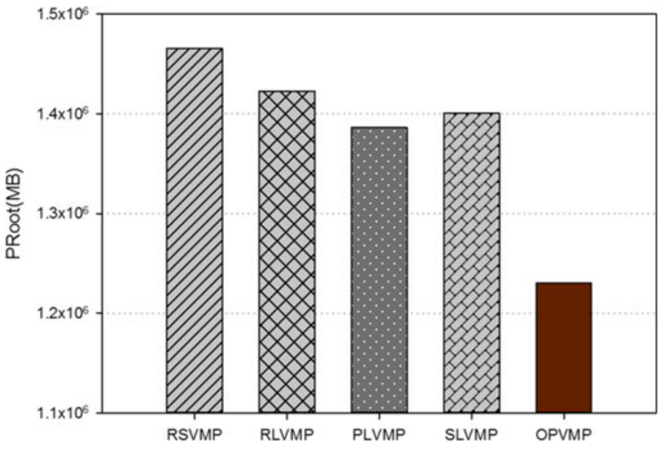

Fig. 4. The performance of root layer network resource consumption (MB).

- *PEdge:* The total size of network packet that has been transferred by the edge layer switches. $PEdge$ can be calculated as follows:

$$PEdge = \sum_i w_e \times size(pkt_i), \tag{23}$$

where w_e denotes the frequency with which packet pkt_i has been transferred by the edge switches.
- *PTotal:* The total size of all packet that has been transferred by the all switches, which can be calculated as follows:

$$PTotal = PRoot + PAgg + PEdge, \tag{24}$$

where w_r denotes the frequency with which packet pkt_i has been transferred by the root switches.

5.2 Performance Evaluations

The experiment involved 20 services, each of which involved 50 primary VMs and 40 backup VMs. Two hundred VM failure events were triggered. Four-thousand data processing tasks were generated. The data size of each task is 300 MB and the task size is set as 10 minutes. The task arrival rate of each service is 200 per hour. The performance of all approaches was studied. Fig. 3 illustrates the performance of $TDelay$. Figs. 4, 5, 6, and 7 present the performance of network resource consumption. Figs. 4, 5, and 6 provide the results of $PRoot$, $PAgg$, and $PEdge$, respectively. Fig. 7 depicts the results of $PTotal$. The results demonstrate that:

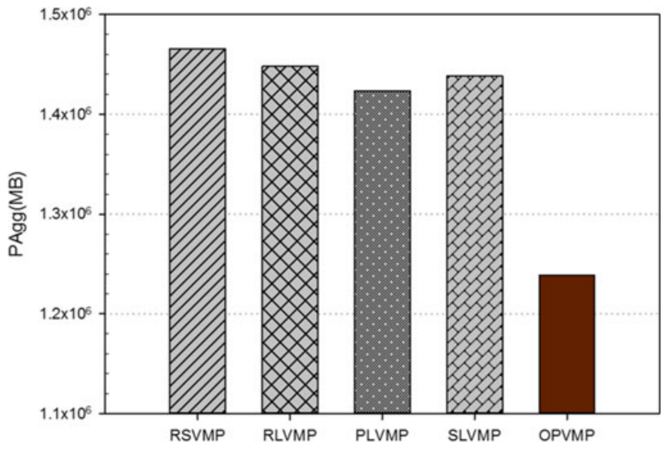

Fig. 5. The performance of aggregation layer network resource consumption (MB).

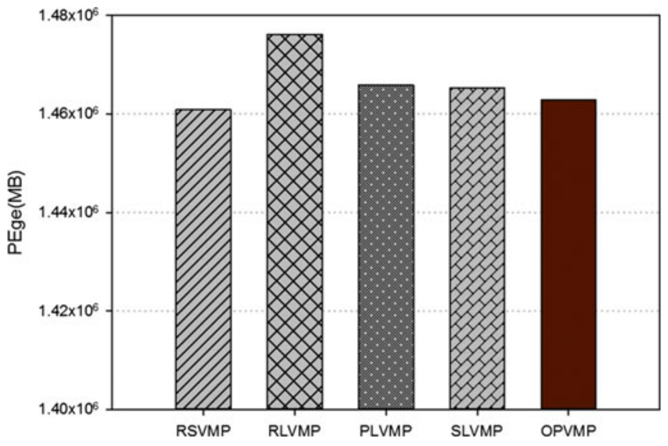

Fig. 6. The performance of edge layer network resource consumption (MB).

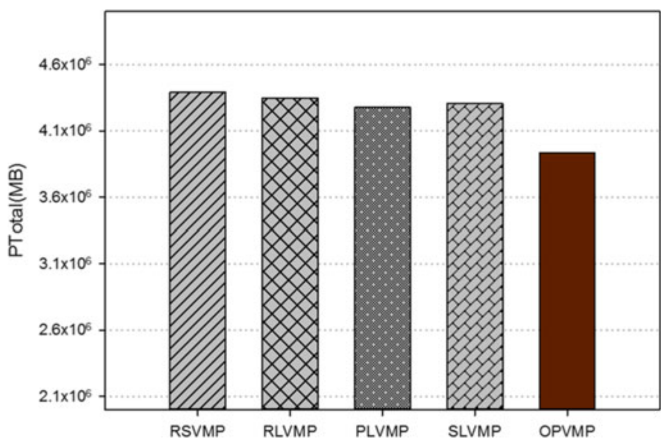

Fig. 7. The performance of total network resource consumption (MB).

- Compared to other approaches, if the data are refetched from the central storage server, more data are processed by the root and the aggregation switches. In addition, the data transfer delay is larger. This is because while retrieving the data from the neighboring host servers is possible, it is unnecessary for the

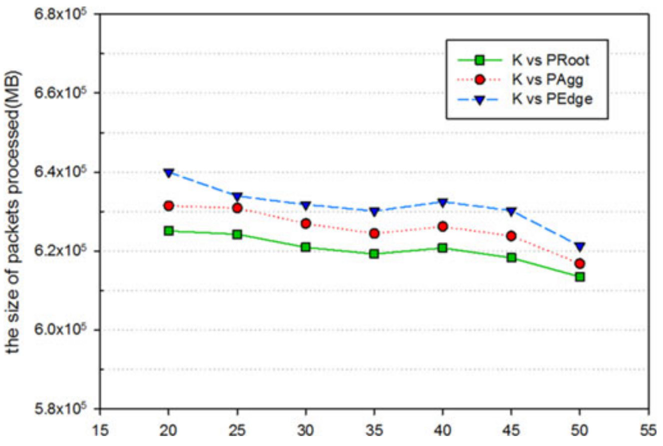

Fig. 8. Impact of parameter k to root layer, aggregation layer, and edge layer network resource consumption (MB), k represents backup VM number.

data traffic transfer to consume upper layer network resources, and the data transfer takes up less time. As our approach OPVMP takes the network topology and the service characteristics into consideration when solving the optimal problem, it consumes the smallest amount of root and aggregation layer network resources.

- Among all the approaches, the edge layer network resource consumption of RSVMP is less than those of other approaches. That's because when the primary VM and the backup VM are in the same pod, the data would transfer through the edge layer switch twice.
- Of all five approaches, our approach consumes the least amount of total network resources, because it takes up less root and aggregation layer network resources than the other approaches.

5.3 Impact of Parameter k

This section contains the result of the study of the impact of parameter k on network resource consumption. The experiment involved 10 services, each of which involved 50 primary VMs. A hundred VM failure events were triggered. Two-thousand data processing tasks were generated. The data size of each task is 300 MB and the task size is set as 10 minutes.

The performance metrics consisted of $PRoot$, $PAgg$, $PEdge$, and $PTotal$. As shown in Fig. 8, caused by some random factor, the network resource consumption increases a little when k increases from 35 to 40. However, the total amount of data that are processed by the root switches, aggregation switches and the edge switches in our approach almost show a decreasing trend when the value of parameter k increases from 20 to 50. Because an increase in k results in an increase in the number of backup VMs for the same service. There is a greater chance that the primary and backup VMs are in the same subnet. Therefore, our approach consumes less upper layer network resources in the recovery stage. Because this reduction occurs as a result of an increase in the value of parameter k, the total network resources also show a decreasing trend in Fig. 9.

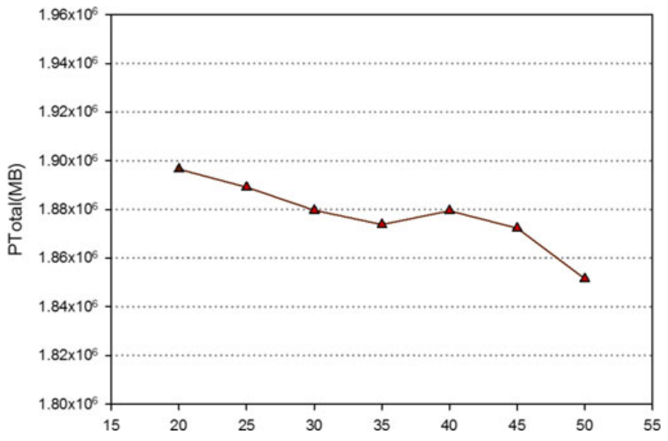

Fig. 9. Impact of parameter k to total network resource consumption, k represents backup VM number.

5.4 Impact of Parameter Task Arrival Rate

We present the impact of task arrival rate on network resource consumption in this section. There are 10 services. Each service has 50 primary VMs and 40 backup VMs. We generate 2,000 data block processing tasks. The data size of each task is 300 MB and the task size is set as 10 minutes. We trigger 100 virtual machine failure events. The task arrival rate of all services increases from 40 to 200 per hour. To show the impact of parameter arrival rate, we calculate the network resource consumption difference between our approach and RSVMP. The performance metrics include: root-layer, aggregation-layer, edge-layer, and total network resource consumption difference between our approach and RSVMP.

As shown in Fig. 10, the root-layer, the aggregation-layer and the edge-layer network resource consumption difference increases with the increase of task arrival rate. As shown in Fig. 11, the total network resource consumption also increases with the increase of task arrival rate. When the task arrival rate increases, the task waiting queue of each VM becomes longer. A failure event will affect more tasks and the total re-fetched data increase. Therefore, our approach can save more upper layer network resource comparing to RSVMP, and the network resource consumption difference increases.

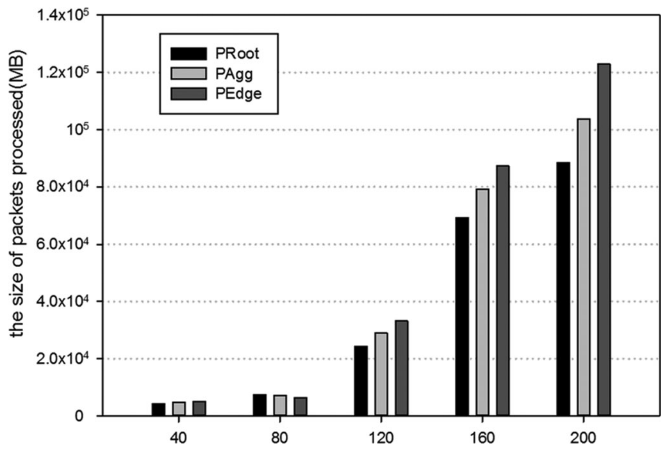

Fig. 10. Impact of task arrival rate to root layer, aggregation layer, and edge layer network resource consumption (MB). The task arrival rate of each service increases from 40 to 200 per hour.

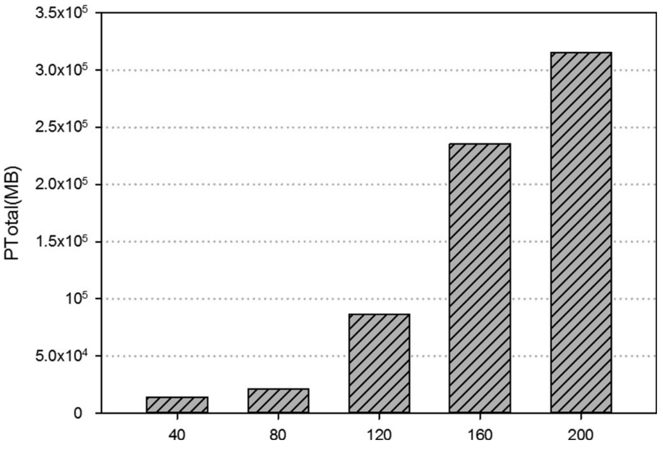

Fig. 11. Impact of task arrival rate to total network resource consumption. The task arrival rate of each service increases from 40 to 200 per hour in our experiment.

6 Conclusions and Future Work

This paper aims at enhancing the reliability of server-based cloud services whose fault-tolerance level can be measured and assured in terms of the replication-based k-fault-tolerance metric. It proposes a novel network-topology-aware redundant VM placement approach to minimizing the consumption of network resources when primary VM failures need be recovered by backup VMs under the k-fault-tolerance constraints. The proposed approach is a three-step process: host server selection, optimal redundant VM placement, and recovery strategy decision. By exploiting the characteristics of the datacenter network, a heuristic algorithm capable of efficiently selecting appropriate host servers and determining the optimal VM placement strategy is presented. Finally, the recovery strategy decision problem is formulated as a maximum weight matching in bipartite graphs problem. An optimal algorithm is presented to solve the problem. The experimental evaluation results show that the proposed approach consumes less network resources than four other representative approaches.

Our future work includes: (1) reducing the complexity of our approach based on some probabilistic analysis, (2) trading off between the effect of the edge switch failure and the network resource saving by adopting a fault avoidance approach, and (3) considering the reliability problem for a complex cloud workflow service.

Acknowledgments

The work presented in this study is supported by NSFC (61472047), NSFC (61272521), Guangdong Natural Science Foundation (Project No. 2014A030313151), and the Fundamental Research Funds for the Central Universities. Shangguang Wang is the corresponding author.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. (2009, Feb.) Above the clouds: A Berkeley view of cloud computing [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[2] W. Voorsluys, J. Broberg, and R. Buyya, “Introduction to cloud computing,” in *Cloud Computing: Principles and Paradigms*. New York, NY, USA: Wiley, 2011, pp. 3–37.

[3] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, “Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility,” *Future Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, 2009.

[4] B. P. Rimal, E. Choi, and I. Lumb, “A taxonomy and survey of cloud computing systems,” in *Proc. 5th Int. Joint Conf. INC, IMS and IDC*, 2009, pp. 44–51.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica et al., “A view of cloud computing,” *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[6] Z. Xia, X. Wang, X. Sun, and Q. Wang, “A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 340–352, Feb. 1, 2016.

[7] M. A Vouk, “Cloud computing—Issues, research and implementations,” *J. Comput. Inf. Technol.*, vol. 16, no. 4, pp. 235–246, 2008.

[8] Y. Ren, J. Shen, J. Wang, J. Han, and S. Lee, “Mutual verifiable provable data auditing in public cloud storage,” *J. Internet Technol.*, vol. 16, no. 2, pp. 317–323, 2015.

[9] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, “Cloud computing‡the business perspective,” *Decision Support Syst.*, vol. 51, no. 1, pp. 176–189, 2011.

[10] M. Melo, P. Maciel, J. Araujo, R. Matos, and C. Araujo, “Availability study on cloud computing environments: Live migration as a rejuvenation mechanism,” in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2013, pp. 1–6.

[11] R. Jhawar, V. Piuri, and M. Santambrogio, “Fault tolerance management in cloud computing: A system-level perspective,” *IEEE Syst. J.*, vol. 7, no. 2, pp. 288–297, Jun. 2013.

[12] (2013). Introduction to designing reliable cloud services. White Paper of Microsoft [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=34683.

[13] E. Bauer and R. Adams, *Reliability and Availability of Cloud Computing*. New York, NY, USA: Wiley, 2012.

[14] M. R. Lyu, et al., *Handbook of Software Reliability Engineering*, vol. 222. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996.

[15] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: State-of-the-art and research challenges,” *J. Internet Services Appl.*, vol. 1, no. 1, pp. 7–18, 2010.

[16] Y.-S. Dai, B. Yang, J. Dongarra, and G. Zhang, “Cloud service reliability: Modeling and analysis,” in *Proc. 15th IEEE Pacific Rim Int. Symp. Dependable Comput.*, 2009, pp. 1–17.

[17] M. Schwarzkopf, D. G. Murray, and S. Hand. (2012). The seven deadly sins of cloud computing research. *Proc. 4th USENIX Conf. Hot Topics Cloud Comput.*, pp. 1–5 [Online]. Available: https://www.usenix.org/conference/hotcloud12/seven-deadly-sins-cloud-co mputing-research

[18] W. Zhao, P. Melliar-Smith, and L. Moser, “Fault tolerance middleware for cloud computing,” in *Proc. IEEE 3rd Int. Conf. Cloud Comput.*, Jul. 2010, pp. 67–74.

[19] I. Goiri, F. Julia, J. Guitart, and J. Torres, “Checkpoint-based fault-tolerant infrastructure for virtualized service providers,” in *Proc. IEEE Netw. Operations Manage. Symp.*, Apr. 2010, pp. 455–462.

[20] N. Limrungsi, J. Zhao, Y. Xiang, T. Lan, H. Huang, and S. Subramaniam, “Providing reliability as an elastic service in cloud computing,” in *Proc. IEEE Int. Conf. Commun.*, Jun. 2012, pp. 2912–2917.

[21] J. Xu, J. Tang, K. Kwiat, W. Zhang, and G. Xue, “Survivable virtual infrastructure mapping in virtualized data centers,” in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, Jun. 2012, pp. 196–203.

[22] Z. Zheng, Y. Zhang, and M. Lyu, “Cloudrank: A qos-driven component ranking framework for cloud computing,” in *Proc. 29th IEEE Symp. Reliable Distrib. Syst.*, Oct 2010, pp. 184–193.

[23] Z. Zheng, T. Zhou, M. Lyu, and I. King, “FTCloud: A component ranking framework for fault-tolerant cloud applications,” in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, Nov. 2010, pp. 398–407.

[24] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, “Performance and availability aware regeneration for cloud based multitier applications,” in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2010, pp. 497–506.

[25] F. Machida, M. Kawato, and Y. Maeno, “Redundant virtual machine placement for fault-tolerant consolidated server clusters,” in *Proc. IEEE Netw. Operations Manage. Symp.*, Apr. 2010, pp. 32–39.

[26] A. S. Tanenbaum and M. Van Steen, *Distributed Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2007.

[27] A. Zhou, S. Wang, C. Yang, L. Sun, Q. Sun, and F. Yang, “FTCloudsim: Support for cloud service reliability enhancement simulation,” *Int. J. Web Grid Services*, vol. 11, no. 4, pp. 347–361, 2015.

[28] A. Zhou, S. Wang, Q. Sun, H. Zou, and F. Yang, "FTCloudsim: A simulation tool for cloud service reliability enhancement mechanisms," in *Proc. Demo Poster Track ACM/IFIP/USENIX Int. Middleware Conf.*, 2013, pp. 1–2.

[29] M. Zhang, H. Jin, X. Shi, and S. Wu, "VirtCFT: A transparent VM-level fault-tolerant system for virtual clusters," in *Proc. IEEE 16th Int. Conf. Parallel Distrib. Syst.*, Dec. 2010, pp. 147–154.

[30] Z. Zheng, T. Zhou, M. Lyu, and I. King, "Component ranking for fault-tolerant cloud applications," *IEEE Trans. Services Comput.*, vol. 5, no. 4, pp. 540–550, 4th Quarter 2012.

[31] (2011). "US government cloud computing technology roadmap volume I high-priority requirements to further USG agency cloud computing adoption," http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-293.pdf

[32] A. L. M. Al-Fares and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 63–74, 2008.

[33] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, 2009.

[34] K. Bilal, M. Manzano, S. Khan, E. Calle, K. Li, and A. Zomaya, "On the characterization of the structural robustness of data center networks," *IEEE Trans. Cloud Comput.*, vol. 1, no. 1, p. 1, Jan.–Jun., 2013.

[35] (2007). Cisco data center infrastructure 2.5 design guide. Cisco Systems, Inc., San Jose, CA 95134-1706, USA [Online]. Available: http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf

[36] A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *Proc. ACM/IEEE Conf. Supercomput.*, 2008, p. 53.

[37] J. E. Hopcroft and R. M. Karp, "An (n5/2) algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, no. 4, pp. 225–231, 1973.

[38] R. E. Burkard and E. Cela, *Linear Assignment Problems and Extensions.* New York, NY, USA: Springer, 1999.

[39] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw.: Practice Experience*, vol. 41, no. 1, pp. 23–50, 2011.

Ao Zhou received the PhD degree in computer science from Beijing University of Posts and Telecommunications of China in 2015. She is an assistant professor at the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. Her research interests include cloud computing and service reliability.

Shangguang Wang received the PhD degree in computer science from Beijing University of Posts and Telecommunications of China in 2011. He is an associate professor at the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. His PhD thesis received the outstanding doctoral dissertation by BUPT in 2012. His research interests include service computing, cloud services, and QoS management. He is a member of the IEEE.

Bo Cheng received the PhD degree in computer science and technology from the University of Electronics Science and Technology of China in 2006. His research interests include multimedia communications and services computing. He is currently an associate professor in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. He is a member of the IEEE.

Zibin Zheng is an associate professor at Sun Yat-sen University, Guangzhou, China. He received Outstanding PhD Thesis Award of The Chinese University of Hong Kong at 2012, ACM SIGSOFT Distinguished Paper Award at ICSE2010, Best Student Paper Award at ICWS2010, and IBM PhD Fellowship Award at 2010. His research interests include service computing and cloud computing. He is a member of the IEEE.

Fangchun Yang received the 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 six 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. He is a senior member of the IEEE.

Rong N. Chang received the PhD degree in computer science and engineering from the University of Michigan in 1990. He is with IBM Research leading a global team creating innovative IoT cloud services technologies. He holds 30 + patents and has published 40+ papers. He is a member of IBM Academy of Technology, ACM distinguished engineer, chair of IEEE Computer Society TCSVC, and an associate editor of *IEEE Transactions on Services Computing*. He is a senior member of the IEEE.

Michael R. Lyu is currently a professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include software reliability engineering, distributed systems, service computing, information retrieval, social networks, and machine learning. He has published over 400 refereed journal and conference papers in these areas. He is a fellow of the IEEE and AAAS for his contributions to software reliability engineering and software fault tolerance.

Rajkumar Buyya is serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in cloud computing. He has authored over 450 publications and five text books including *Mastering Cloud Computing* published by McGraw Hill and Elsevier/Morgan Kaufmann, 2013 for Indian and international markets, respectively. Software technologies for Grid and Cloud computing developed under his leadership have gained rapid acceptance and are in use at several academic institutions and commercial enterprises in 40 countries around the world. He has led the establishment and development of key community activities, including serving as the foundation chair in the IEEE Technical Committee on Scalable Computing and five IEEE/ACM conferences. These contributions and international research leadership of him are recognized through the award of "2009 IEEE TCSC Medal for Excellence in Scalable Computing." He is currently serving as a co-editor-in-chief of *Journal of Software: Practice and Experience*, which was established 40 + years ago. He is a fellow of the IEEE, professor of computer science and software engineering, future fellow of the Australian Research Council, and the director in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia.