Network Aware Resource Allocation in Distributed Clouds

Mansoor Alicherry
Bell Labs India, Alcatel-Lucent
Bangalore, India

T.V. Lakshman
Bell Labs, Alcatel-Lucent
New Jersey, USA

***Abstract*—We consider resource allocation algorithms for distributed cloud systems, which deploy cloud-computing resources that are geographically distributed over a large number of locations in a wide-area network. This distribution of cloud-computing resources over many locations in the network may be done for several reasons, such as to locate resources closer to users, to reduce bandwidth costs, to increase availability, etc. To get the maximum benefit from a distributed cloud system, we need efficient algorithms for resource allocation which minimize communication costs and latency. In this paper, we develop efficient resource allocation algorithms for use in distributed clouds. Our contributions are as follows: Assuming that users specify their resource needs, such as the number of virtual machines needed for a large computational task, we develop an efficient 2-approximation algorithm for the optimal selection of data centers in the distributed cloud. Our objective is to minimize the maximum distance, or latency, between the selected data centers. Next, we consider use of a similar algorithm to select, within each data center, the racks and servers where the requested virtual machines for the task will be located. Since the network inside a data center is structured and typically a tree, we make use of this structure to develop an optimal algorithm for rack and server selection. Finally, we develop a heuristic for partitioning the requested resources for the task amongst the chosen data centers and racks. We use simulations to evaluate the performance of our algorithms over example distributed cloud systems and find that our algorithms provide significant gains over other simpler allocation algorithms.**

I. Introduction

A key function performed by cloud management and automation software is resource allocation. Typically, user requests for a service hosted in the cloud require the allocation of virtual machines (VMs) in the cloud data centers, to meet the requested service's computational needs. A basic example is a request for infrastructure-as-a-service where the user may explicitly request a number of VMs and also the desired connectivity between the VMs. The cloud management and automation software then identifies the right physical resources for each of the requested VMs and allocates them. For this the resource allocator maintains a continually updated view of all the resources available in the different data centers, their capabilities, and their current and future allocations. Resource allocation algorithms, used by the cloud automation software, have a very high impact on the performance of users' applications as well as on the ability of the data center to accommodate the maximal number of user requests. This is particularly true for distributed cloud systems where the resource allocation algorithms may have to split the resource allocation for a user-request over multiple relatively smaller data centers.

When a user makes a request to run an application in the data center, the request specifies the needed number of VMs and the desired communication requirement among those VMs. The cloud automation software's objective is to choose data-centers and racks for assignment of the requested VMs so as to reduce the overall resource usage and provide the best performance to the application. These two goals are complementary since the best performance is obtained when all the VMs are located in the same rack and this also uses the least amount of inter-rack traffic. When a user request arrives, a rack may not exist that has sufficient residual capacity to accommodate all the requested VMs. Hence, VMs may be allocated from amongst multiple racks. One goal of the cloud automation software is to minimize the inter-rack communication.

Resource allocation algorithms that handle a variety of usage scenarios are needed. Typically, user requests (jobs) may arrive and leave at any time. A user may also make requests for more resources as time progresses. To handle these, online algorithms are needed. However, When jobs are scheduled in an online manner, the data center capacity may get fragmented. This may lead to VMs of the applications being scheduled in multiple racks, reducing the utilization as well as affecting the performance of the applications. The utilization of the DC and the application performance may be improved by bringing together the VMs that are dispersed. Periodically, cloud automation can performs defragmentation by running live migration to bring those VMs together. This need to be done in such a way that the resulting data center utilization is improved. The defragmentation algorithm also needs to minimize the number of VM movements.

Another resource allocation problem in cloud automation arises when requests come in batches. This can happen when customers request for resources that need to be scheduled in future. Here the cloud automation tool has opportunity to perform optimization across the jobs, called batch optimization. It may also be necessary to perform admission control of the new jobs, based on the commitments already made.

In this paper, our focus is on resource allocation problems in distributed cloud systems. In these systems, the cloud resources are geographically distributed and interconnected over a wide-area network. Because of the distribution and the relatively smaller size of the data centers, it is possible that a single user request may have its resources allocated from amongst multiple data centers. Consequently, the latency in communication between the different centers is far more significant than in the case of centralized cloud architectures where the cloud resources are concentrated inside a few large data centers. We develop resource allocation algorithms for distributed cloud systems and a primary objective is to minimize the maximum latency in communication between the virtual machines allocated for a user request. We pick this objective so as to reduce the possibility of tasks running on distant pairs of virtual machines which will lead to large communication latencies and hence delay overall completion times for the user request.

II. System Architecture

A. Distributed Cloud

Distributed cloud architectures [16] consist of a large number of small sized data centers distributed across a geographic area. This architecture is appealing to network service providers who already have the necessary distributed facilities (such as central offices that are geographically dispersed and close to users) for deploying a large number of distributed data centers interconnected by high-speed networks. Distributed cloud architectures can provide several benefits over the traditional centralized cloud architectures, where large datacenters are placed at a few locations. In a distributed datacenters, customer requests can be serviced from locations closest to them. This helps reduce network capacity needs, for high-bandwidth applications, which constitute a significant cost when accessing centralized datacenters [6]. Distributing the datacenters also reduces the latency of access compared to traditional datacenters. In fact, the access latency of the traditional datacenter may have large variation due to the long path lengths and going through multiple service providers [12]. Figure 1 illustrates a distributed cloud.

Though several novel intra-data-center network architectures have been recently been proposed in the research literature [7], [8], [14], current intra-data-center network deployments are typically organized in a hierarchical manner (Figure 2). As the figure shows, each rack contains a fixed number of blade servers. Each blade server has a few processors each having several processing cores. The VMs that are running in different cores of the same blade can communicate directly without going through any external switch. Machines in different blade servers that are part of the same rack communicate using a top-of-the-rack (TOR) switch that is attached to the rack. Two racks communicate using aggregator switches. Hence, machines in blade servers located in adjacent racks communicate using a path that consists of the TOR switch of the source rack, aggregator switch and the TOR switch of the destination rack. If the racks are located farther apart, there may be multiple levels of aggregate switches. Hence, the latency of the communication for the VMs of an application depends on the location of the physical machines on which they are scheduled.

Current data center networks are designed with the assumption of locality of communication. That is most of the communication is assumed to be amongst machines in the same rack. Aggregate switches can carry only a fraction of a rack's network capacity. As the distance between the machines increase, the available bandwidth between them decreases. Hence, the available bandwidth for the VMs of the application depends on the physical machines that they are assigned to. Furthermore, the overall efficiency of the datacenter also depends on this assignment. The number of requests that the cloud management and automation software will be able to admit will depend on the available bandwidth of the datacenter.

B. Cloud Management and Automation Architecture

Figure 3 shows the high-level architecture of the cloud management and automation system. Users request services from the cloud. Each user request consists of a number of VMs that need to be allocated and the communication requirements among the VMs. It is possible that the user many not have a priori knowledge of the communication requirements amongst the VMs. In that case, the cloud automation system may need to do an initial assignment based on worst-case assumptions and re-optimize based on actual measurements. For our algorithms, we assume that knowledge of the communication requirements amongst the VMs is known and made available for resource allocation computations. The user may also specify additional constraints to meet fault tolerance and elasticity needs. For example, the user may specify a limit on the number of VMs that may be placed at a datacenter for fault tolerance purposes. The system may also impose a restriction that a data center needs to host at least a certain minimum number of VMs to reduce the inter-data-center traffic.

The cloud automation software computes a placement of VMs for the user request. The output contains a mapping of VMs to the physical resources. This mapping specifies the datacenter, rack, blade, and the CPU where the VM will be scheduled. It also specifies any network configurations that need to be performed.

To perform its assignment function, the cloud automation software interacts with the network management system (NMS) and the local cloud management system (CMS) in the data centers. NMS provides a view of the current network between the datacenters. Cloud automation uses it to infer the cost and bandwidth availability for inter-data-center traffic. CMS provided the view of the individual data centers. They include the availability of network, storage and compute resources in the datacenter. Cloud automation software provides the placement to the CMS and NMS to allocate resources for the user request.

The cloud automation software module has two main functionalities: Keeping track of resource usage, and optimized assignment of user requests. It maintains the availability and usage of networking and compute resources in its database.

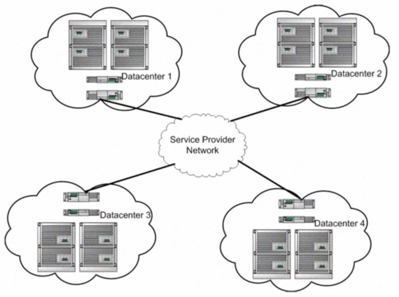

Fig. 1. A distributed cloud architecture

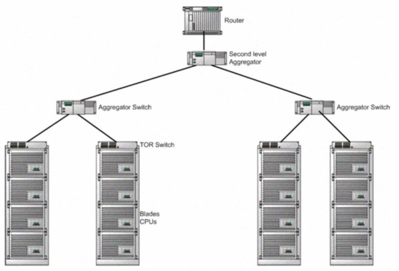

Fig. 2. A typical datacenter in the distributed cloud

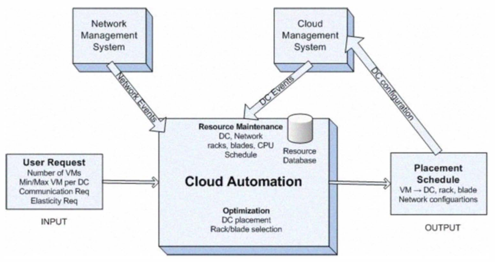

Fig. 3. Architecture of a cloud resource allocation system

This database is kept current by getting information from the NMS and CMS.

The assignment of resources to a user request consists of identifying the datacenters and the machines (racks and blades) where the user's request will be run. The goal here is to reduce the inter-datacenter and intra-datacenter traffic as well as to minimize the path length of the data packets to improve the application performance. The assignment is performed in four steps:

1) **Datacenter Selection:** Identify the datacenters to place the user request. Based on user constraints and availability the request may be placed in more than one datacenter. We identify a subset of the datacenters that satisfy the resource needs and minimize the length of the paths between the datacenters.. In particular, we are interested in minimizing the length of longest path or more generally in minimizing the largest communication latency. This is because VMs that incur large communication latency are likely to lag in their task completion times and hence increase overall completion times for the user request.
2) **Request partitioning across datacenters:** After identifying the set of datacenters suitable for assigning VMs for the current user request, we need to determine the data-center assignment for each individual VM. For this assignment, our objective is to minimize the inter-datacenter traffic. The requests might also have specified additional constraints on placing different types of VMs on different data centers. The cloud automation software adheres to these constraints during placement.
3) **Rack, blade and processor selection:** Here we identify the physical compute resources in each of the selected datacenters. The goal again is to identify machines that have low inter-rack traffic.
4) **VM placement:** In this step, we assign individual VMs to the physical resource (i.e rack, blade, processor) identified in step 3. In this step we try to minimize the inter-rack traffic between the VMs.

In the rest of the paper, we develop optimization algorithms for each of these resource allocation steps. In both Steps 2 and 4, we partition the VMs such that it reduces the inter-partition traffic and the algorithms used in these steps are similar in objective. Solving steps 2 and 4 as separate problems, instead of combining them together reduces the problem size as well as enables adding additional user constraints to the scheduling.

III. Data center selection

In a distributed cloud environment, datacenters are placed at multiple geographic locations. The first step in servicing a user request is selection of the right datacenters to place the VMs. A single datacenter may not have enough capacity to host all the VMs of the user. Even if there is enough capacity a datacenter, the user may not want to have all the VMs hosted in one datacenter, in order to guard against complete failure if the data center has a service outage. The cloud automation software needs to select the datacenters for VM deployment that meet user constraints, optimize network use, and maximize the application performance. In this section, we present an algorithm that selects a subset of datacenters for placing the VMs of a user request such that it minimizes the maximum distance (or the hop count) between any two datacenters. We use this objective since we want to avoid the chances of certain VMs strongly lagging others in task completion due to large communication latencies between some pairs of VMs (lagging VMs result in increased completion times for user requests). The algorithm that we present can also handle additional user constraints like minimum and maximum number of VMs placed in any datacenter.

Datacenter selection problem may be viewed as a subgraph selection problem, which we call MINDIAMETER. We are given a complete graph $G = (V, E, w, l)$. The vertices V, represents the datacenters, and weights w on them denotes the number of available VMs in that datacenter. The edges E represents the path between the datacenters and the labels (length) l on them denotes the distance, number of hops or latency of the shortest path. We use the term lengths and weights to denote the weights on the edge and the vertices respectively. If the user request has any constraint on the maximum number of VMs that may be placed at a datacenter, then the weights of the vertices are capped to that number. Similarly, if the request has constraint on minimum number of VMs on a datacenter, then the vertices with fewer weights are removed from the graph.

Let s be the number of VMs requested by the user. Data center selection problem MINDIAMETER(s) corresponds to finding a subgraph of G, whose sum of weights is at least s and with minimum diameter (i.e maximum length of any shortest between the vertices). Since the original graph is a complete graph, the subgraph induced by the selected vertices

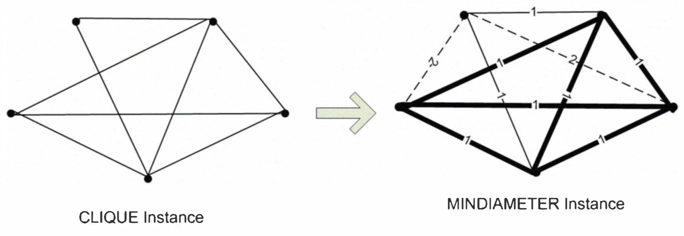

Fig. 4. Reduction from MAXCLIQUE to MINDIAMETER problem

is also complete. Hence, our goal is to find such a subgraph whose length of the longest edge is minimum.

A. Hardness of approximation

MINDIAMETER problem is NP-hard and cannot be approximated within $2-\epsilon$ for any $\epsilon > 0$. We can reduce it from MAXCLIQUE problem [4], where the problem is to find a clique of maximum size. The reduction is as follows. Given an instance $G = (V, E)$ of MAXCLIQUE problem, we create an instance of MINDIAMETER problem. We create a new complete graph $G' = (V', E', w, l)$. The vertices of G' are same as G, and has weight 1. If there is an edge between two vertices u and v in G, then the length of the edge between the corresponding vertices of G' is 1. Otherwise, the length of the edge between the vertices is 2. The edges in this graph satisfy the triangle inequality. A subgraph of G' has a diameter 1, if and only if the corresponding subgraph in G is a clique. This is because, if G has a clique of size k, then we can take the corresponding vertices of G' to form a subgraph of weight k and diameter 1. Similarly, if G' has a subgraph of weight k and diameter 1, then the corresponding vertices of G forms a clique of size k. Hence, we can find the solution to MAXCLIQUE by finding maximum $s \in \{n, n-1, \ldots, 1\}$ for which there exist a subgraph of size s and diameter 1.

Example: Figure 4 gives an instance of the MAXCLIQUE problem and the equivalent MINDIAMETER problem. The labels on the edge show the link costs. Dotted lines are the new edges added to make the graph a complete graph. Thick lines shows the edges of the MINDIAMETER solution, where the diameter of the graph is 1. This is also the maximum clique in the original graph.

Lemma 1: MINDIAMETER problem cannot be approximated in polynomial time within a factor $2-\epsilon$ for any $\epsilon > 0$, unless P = NP.

Proof: If there is $2-\epsilon$ approximation algorithm for MINDIAMETER, we can solve the MAXCLIQUE problem as follows. To find a clique of size k, create a MINDIAMETER problem with weight k using the above reduction. If there is a clique of size k, then there exists a subgraph for MINDIAMETER problem of diameter 1. If there is no such clique, then the diameter of the MINDIAMETER problem is at least 2. Hence, if a $2-\epsilon$ algorithm for MINDIAMETER returns a subgraph whose diameter is less than 2, then there is clique of size k in the original graph.

B. Approximation algorithm

We describe an algorithm for minimum diameter subgraph problem that gives the best approximation guarantees. The diameter of the subgraph output by the algorithm is at most twice the diameter of the optimal subgraph. We assume triangle inequality for the edge weights in the graph. Triangle inequality is expected to be generally valid in our problem settings since the length of the edges corresponds to path length between the corresponding data centers. If there is a triangle inequality violation between three data centers, then we can conform to the triangle inequality by forcing the longer edge to take a path formed by the shorter edges.

Algorithm 1 FindMinStar(G, v, s)

```
1:  Input: $G = (V, E, w, l)$: Complete graph with vertex
    weight and edge lengths
    $v$: Starting vertex
    $s$: Required weight of the subgraph
2:  Output: Subgraph $G' = (V', E')$ of weight at least $s$
    formed by $v$ and its closest neighbors.
3:  $V' \leftarrow \{v\}$, $E' \leftarrow \phi$, $size \leftarrow w(v)$
4:  Let $u_1, u_2, \ldots, u_{n-1}$ be the vertices of $G$ sorted in in-
    creasing order of length to $v$. (i.e. $l(v, u_i) \leq l(v, u_{i+1})$)
5:  $i \leftarrow 1$
6:  $diameter \leftarrow 0$
7:  while $size < s$ and $i < n$ do
8:      $V' \leftarrow V' \cup \{u_i\}$
9:      $size \leftarrow size + w(u_i)$
10:     $i \leftarrow i + 1$
11:     $diameter \leftarrow max(diameter, \{l(v', u_i) : v' \in V'\})$
12:     $E' \leftarrow E' \cup \{(v', u_i) : v' \in V'\}$
13: end while
14: if $size < s$ then
15:     No subgraph of size $s$ exist in $G$. Return NULL
16: end if
17: return $G' = (V', E')$ and $diameter$
```

The algorithm $FindMinStar(G, v, s)$ in Algorithm 1 finds a subgraph of weight at least s that includes vertex v. It finds a star topology centered at vertex v by adding nodes in increasing order of length to v, until the weight of the star topology is at least s. The edges of the subgraph are the edges induced by the nodes in the star. The algorithm also computes the diameter of the resulting subgraph. This is done by maintaining the diameter as the nodes are added. When a node is added, the diameter of the subgraph can change only if the length of the edges induced by that node is greater than the current diameter.

Algorithm $MinDiameterGraph(G, s)$ in Algorithm 2 finds a subgraph of weight at least s, by invoking $FindMinStar$ for each of the vertices, and selecting the one with smallest diameter.

C. Analysis

In this section, we show that the algorithm presented above is a 2-approximation algorithm.

Algorithm 2 MinDiameterGraph(G, s)

```
1:  Input: $G = (V, E, w, l)$ : Complete graph with vertex
    weight and edge lengths.
    $s$: Weight of the subgraph
2:  Output: Subgraph $G' = (V', E')$ of weight at least $s$.
3:  $mindiameter \leftarrow \infty$
4:  for all vertex $v \in V$ do
5:      $G'' \leftarrow FindMinStar(G, v, s)$
6:      if $diameter(G'') < mindiameter$ then
7:          $mindiameter \leftarrow diameter(G'')$
8:          $G' \leftarrow G''$
9:      end if
10: end for
11: return $G'$ and $mindiameter$
```

Lemma 2: $FindMinStar(G, v, s)$ finds the subgraph of weight at least s, whose length of any edge incident on v is minimum.

Proof: $FindMinStar(G, v, s)$ algorithm first adds vertex v to the subgraph. Then it adds the vertices adjacent to v, in the increasing order of edge weights, until the sum of the weight of the vertices in the subgraph is s. Hence, the algorithm finds a subgraph with smallest edge lengths for the edges incident on v.

Theorem 3: $MinDiameterGraph(G, s)$ finds a subgraph of weight at least s, whose diameter is at most twice the optimum.

Proof: $MinDiameterGraph(G, s)$ invokes $FindMinStar$ algorithm for each of the vertices of G and selects the one with minimum diameter. Let v' be the node for which the diameter of the graph G' returned by $FindMinStar$ was minimum. Let l' be the length of the longest edge incident on v' in the subgraph G'. Since the edge lengths follow triangle inequality and G' is a complete graph, the length of any edge in G' is at most $2l'$. Hence, diameter of G' is at most $2l'$.

Let G_{opt} be the optimal solution to MINDIAMETER(s). Let l'' be the longest edge of G_{opt}. Let u'' and v'' be the end points of l''. Since the edges of G_{opt} satisfies triangle inequality the shortest distance in G_{opt} between u'' and v'' is l''. Hence the diameter of G_{opt} is at least l''. Now consider the graph G'' returned by $FindMinStar(G, s, v'')$ while running the $MinDiameterGraph(G, s)$. According to the lemma 2, the lengths of edges incident on v'' on G'' is at most l''. Since the diameter of G'' is at most $2l''$, the diameter of the subgraph returned by $MinDiameterGraph(i.e.G'')$ is at most $2l''$. Hence, our algorithm is a 2-approximation algorithm.

Running time: Algorithm $FindMinstar$ needs to sort the lengths of edges incident on a node, which takes $O(n \log n)$, where n is number of datacenters. While loop in the algorithm may be executed once per node. Computing diameter takes $O(n^2)$ as there are n^2 edges. Hence, the worst case running time of $FindMinstar$ is $O(n^2)$. $MinDiameterGraph$ invokes $FindMinstar$ n times, one for each node giving the worst case complexity $O(n^3)$.

IV. Machine selection inside datacenter

Once the datacenters have been identified to place the user request, we need to identify the physical resources inside the chosen datacenters where the VMs could be assigned. A datacenter may have more available VMs than the one requested by the user. In those cases, choosing the right set of physical resources to run the VMs can help to improve both datacenter utilization as well as performance of the user application.

The goal of the machine selection is to find the machines that reduce the inter-rack communication and avoid long paths for the communication. In a data center that consists of 100s of racks, it is important that VMs of an application be scheduled in racks that are physically located close to each other. Otherwise, the communication may have to go through multiple aggregate switches leading to large latency and resource (bandwidth) usage.

For the machine selection, we could use the same algorithm as the datacenter selection to get a 2-approximation algorithm to minimize the maximum distance between the machines. Here the vertices of the graph would represent the racks whose weights would represent the number of available VMs, and the edges lengths would represent inter-rack distance.

However, as noted earlier, since the datacenter topology is often organized as hierarchical network, we can solve the machine selection problem for this case optimally in the min-max sense. Here, the topology may be considered as a tree topology, where the root represents the core level switches, its children represents top-level aggregate switches, grandchildren represents second-level aggregate switches and so on. Finally, the leaf nodes represent the racks. In this tree, all the leaves are at the same level (i.e. distance from the root). We also add labels on the leaves to represent the number of available VMs on each rack. It is also possible to extend this tree to blade level, where the leaf nodes represent the blades and their parents represents the racks.

For the machine selection problem, we minimize the maximum communication distance between any two VMs. Here the problem can be translated into finding a rooted sub tree of minimal height whose sum of the labels on the leaves is at least the target number of VMs required. Like the datacenter selection problem, we may include additional constraints like the maximum and minimum VMs that are placed per rack. We can include these constraints by changing the weight of the leaves accordingly.

Let s be the number of VMs that needs to be placed at a datacenter. Let T be the tree representation of the datacenter compute and networking resource. We associate two variables with each node in T; $weight(v)$ represents number of available VMs rooted at node v and $height(v)$ represents the height of the node. Before running the algorithm the weight variables are set only for the leaf nodes.

Algorithm $FindMinHeightTree(T, r, s)$ in Algorithm 3 finds a subtree of the tree rooted at r whose leaf nodes have a cumulative weight of at least s and of minimum height. The algorithm performs the post-order traversal of the tree, and maintains the height and weight of each node as well as root of the minimum height subtree with weight at least s.

Running time: The algorithm has the same complexity as tree traversal: $O(n)$.

Algorithm 3 FindMinHeightTree(T, r, s)

```
1: Input: $T$ : Tree representation of the datacenter resource
     $r$ : Root of the subtree where we want to start search
     $s$ : Required weight of the subtree
2: Output: Root of the subtree with weight at least $s$ that
     has minimum height. Algorithm also computes the weight
     and height of the current subtree.
3: if $r$ is a leaf node then
4:   $height(r) \leftarrow 1$
5:   if $weight(r) \geq s$ then
6:     return $r$
7:   else
8:     return NULL
9:   end if
10: end if
11: $height \leftarrow 0$, $weight \leftarrow 0$
12: $minheight \leftarrow \infty$, $mintree \leftarrow NULL$
13: for each vertex $v$ in children($r$) do
14:   $n \leftarrow FindMinHeightTree(T, v, s)$
15:   if $n \neq$ NULL and $minheight < height(n)$ then
16:     $minheight \leftarrow height(n)$
17:     $mintree \leftarrow n$
18:   end if
19:   if $height(v) > height$ then
20:     $height \leftarrow height(v)$
21:   end if
22:   $weight \leftarrow weight + weight(v)$
23: end for
24: $height(r) \leftarrow height + 1$
25: $weight(r) \leftarrow weight$
26: if $mintree =$ NULL and $weight \geq s$ then
27:   return $r$
28: else
29:   return $mintree$
30: end if
```

V. Virtual Machine Placement

In this section, we provide heuristic algorithms for assigning individual VMs to datacenters and to CPUs within the datacenters. This problem is a variant of graph partitioning and k-cut problems (Section VII). Our goal is to device algorithms that can be implemented on cloud automation systems, and does not require expensive computations.

We represent the user request as a graph $G = (V, E)$, where the nodes represent the tasks (VMs) to be placed and the edges represent the communication requirements between them. Our goal is to partition V into disjoint sets $C_1, C_2, \ldots C_m$, such that communication between vertices belonging to different partition is minimized. For ease of representation, we assume symmetric communication. For asymmetric traffic, we take the average of incoming and outgoing traffic in each link.

Each partition of the graph represents the set of VMs that need to be scheduled in the same datacenter (for global or cloud scheduling) or same rack (for inside datacenter scheduling). The size of the each partition needs to be upper-bounded by the number of available VMs in the corresponding datacenter or rack. Unlike the traditional graph partition problem, our problem may not specify the exact number of nodes in each partition; it only specifies the maximum nodes in each of the partition. This is because at any point, there may be more VMs available in the system than the request. The algorithm tries to optimize the communication by scheduling the VMs within the maximum available VMs of each datacenter or rack.

Algorithm 4 Algorithm: FindCluster(G, s)

```
1: Input: $G = (V, E)$ : Input graph with communication
     requirement between the nodes. Weight $w$ of an edge
     represents the communication requirement
     $s$ : Number of nodes in the cluster.
2: Output: Set of nodes $C$ to schedule in the cluster.
3: $v \leftarrow$ vertex with maximum $\sum_{u \in V} w(v, u)$
4: $C \leftarrow \phi$
5: $traff(u) \leftarrow 0$ for all $u \in V$
6: while $s > |C|$ and $s > |V|$ do
7:   $C \leftarrow C \cup \{v\}$
8:   for each vertex $u \in V - C$ adjacent to $v$ do
9:     $traff(u) \leftarrow traff(u) + w(v, u)$
10:  end for
11:  $v \leftarrow \arg\max_{u \in V - E} traff(u)$
12: end while
13: return $C$
```

Algorithm 5 Partition(G, K)

```
1: Input: $G = (V, E)$ : Input graph with communication
     requirement between the nodes.
     $K = k_1, k_2, \ldots k_r$: Size of clusters to partition.
2: Output: A partition of $G$ with components $C_1, C_2, \ldots C_r$
     such that $|C_i| \leq k_i$.
3: Let $k_1, k_2, \ldots k_r$ be in decreasing order
4: $V' = V$
5: for $i$ = 1 to r do
6:   $G' \leftarrow$ subgraph of $G$ induced by vertex set $V'$
7:   $C_i \leftarrow FindCluster(G', k_i)$
8:   $V' \leftarrow V' - C_i$
9: end for
10: return $C_1, C_2, \ldots, C_r$
```

Algorithms 4 and 5 give a heuristic solution for the partition problem. Algorithm $Partition(G, K)$ is given the user request graph G, the maximum number of nodes in each partition $K = k_1, \ldots, k_r$ as input. Set K is derived from the output of algorithms in Sections III and IV.

This algorithm selects datacenters (or racks) in the decreasing order of available capacity and fills as many VMs as possible in those datacenters (or racks) using $FindCluster$. The

$FindCluster$ algorithm selects a VM with maximum amount of incoming/outgoing bandwidth (or number of neighbors) and places it in the datacenter (or rack). This VM is added to a set C, which is the set of scheduled VMs in the datacenter (or rack). Then it considers all the neighbors of C. It schedules the one that has maximum traffic towards/out of C and adds the VM to set C. This process is repeated until all the available VMs of the datacenter (or rack) have exhausted or there are no more VMs to schedule.

The greedy solution may be improved by exchanging certain nodes in the schedule using Kernighan-Lin [11] heuristic or its variants. The idea is to consider pairs of nodes that are present in different partition and checks if interchanging of these nodes improves the solution (i.e. uses less inter-rack bandwidth). We also can check if moving a node from one partition to another that has available capacity improves the solution. The algorithm selects the best move possible and commits that move. This process is repeated until a threshold number of moves have been performed or there is no further improvement in solution.

Running time: In $FindCluster$, while loop is executed one for each node (VM). $traff$ variable may be maintained in an heap. For each node, this variable gets updated at most once for every neighbor, giving the time complexity of $O(n^2 \log n)$.

VI. Simulation Results

In this section, we evaluate the performance of our algorithms. First, we compare our placement algorithm in Section III (*Approx*) with two other algorithms: *Random* and *Greedy*. *Random* algorithm selects a random datacenter and places as many VMs from the request as possible in that datacenter. If there are more VMs in the request than what is available in the selected datacenter, then the algorithm randomly chooses the next datacenter to place the remaining VMs. This process is repeated until all the VMs in the request are placed in some datacenter. *Greedy* algorithm selects the datacenter with maximum number of available VMs. It places as many VMs from the request as possible on that datacenter. If there are remaining VMs in the request to be placed, then the algorithm selects the next datacenter with largest number of available VMs. This process continues until all the VMs are placed.

To measure the performance of the algorithms, we create random topologies and user requests, and measure the maximum distance (diameter) between any two VMs in the placement output by these algorithms. The locations of datacenters are randomly selected from 1000x1000 grid. The distance between these datacenters is the Euclidean distance between the points. There are five different distributed cloud scenarios containing 100, 75, 50, 25 and 10 datacenters. The average number of machines in each of the clouds is the same. Hence, the number of machines in a datacenter is inversely proportional to number of datacenter in the corresponding cloud. Number of machines per datacenter on a 100-datacenter cloud is chosen uniformly random between 50 and 100. For the 50-datacenter cloud the number of machines is uniformly random between 100 and 200, and so on. In each of the experiments below, we report the results as average of 100 runs.

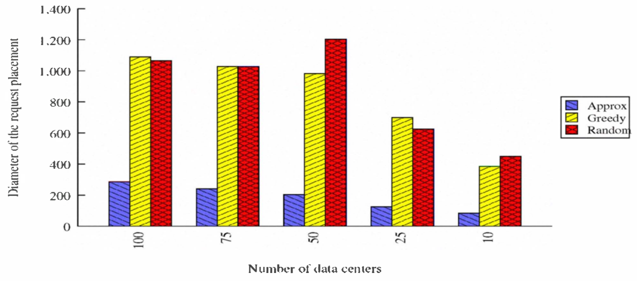

Fig. 5. Diameter of the placement for a request of 1000 VMs

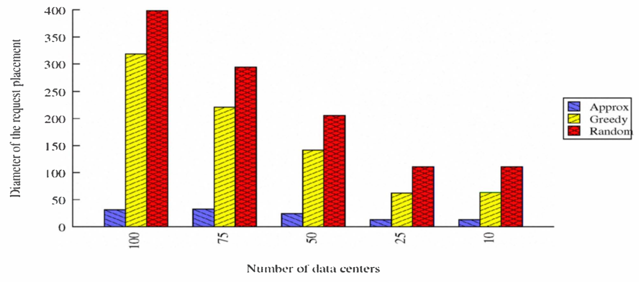

Fig. 6. Diameter for placing 100 requests of 50 to 100 VMs

In the first experiment, we measure the diameter of the placement for a single request of 1000 VMs. Figure 5 shows the results for each of the algorithms. Our approximation algorithm significantly outperforms Greedy and Random in all the distributed cloud scenarios by 79%. Greedy and Random have similar performance. It can also be noted that the diameter decreases as number of datacenters decrease. This is because, number of machines available per datacenter increases as number of datacenter decrease. Hence, a smaller number of datacenters can service the request in dense clouds, reducing the diameter of the placement.

Now we study the cloud systems with a series of user requests. We conduct two sets of experiments and measure the average diameter of placement. In the first set, there are 100 requests for 50 to 100 VMs uniformly distributed. We call it *large requests*. In the second set, there are 500 requests for 10 to 20 VMs. We call it *small requests*. Note that the average number of VMs requested in both the experiments is the same. Figures 6 and 7 give the average diameter for the algorithms for large and small requests respectively. In these experiments greedy performs better than random by 32.6% and 66.5% respectively. Approx performs better than greedy

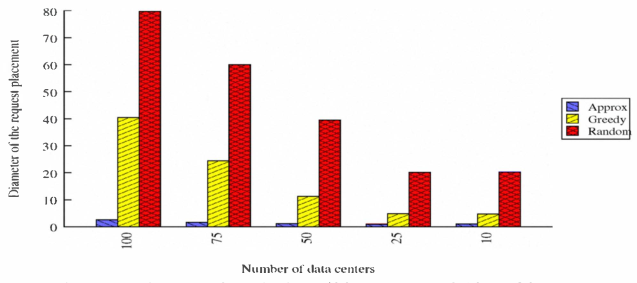

Fig. 7. Diameter for placing 500 requests of 10 to 20 VMs

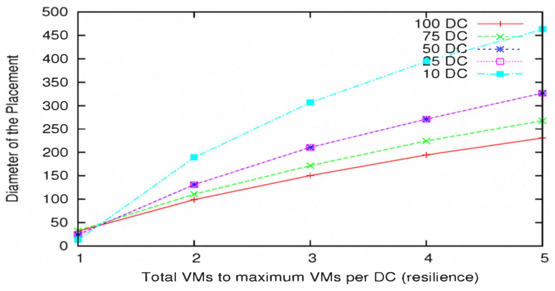

Fig. 8. Diameter for 100 requests of 50-100 VMs for different resilience

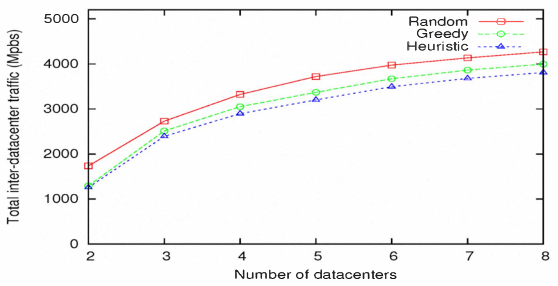

Fig. 10. Inter-datacenter traffic for 100 VMs with excess capacity in DCs

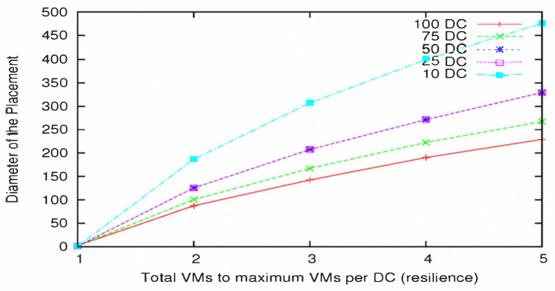

Fig. 9. Diameter for 500 requests of 10-20 VMs for different resilience

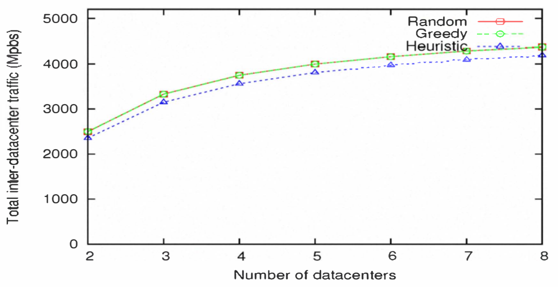

Fig. 11. Inter-datacenter traffic for 100 VMs with no excess capacity in DCs

by 83.4% and 86.4% respectively. For the same algorithm, placement of larger requests requires higher diameter than the smaller requests. This is because, larger requests may have to be placed in multiple datacenters, which increases the diameter of the placement. We also see the trend of decreasing diameter as the number of datacenter in the cloud decreases, as observed in the previous experiment.

Now we study the performance of the cloud systems when user gives additional constraints on the maximum VMs that may be placed at a datacenter. We use the same set of requests as the previous experiment: large and small requests. In addition, user specifies the maximum number of VMs that can be placed at a datacenter. This is specified as the ratio of total number of VMs to maximum VMs in any datacenter, called *resilience*. The request needs to be placed in at least resilience number of datacenters. Since approx performs significantly better than other algorithms, we only present the results for approx. Other algorithms also behaved in a similar manner with larger diameters.

Figures 8 and 9 show the diameter of the placement by approx as a function of resilience for clouds containing different number of datacenters for large and small requests respectively. As observed in the previous experiment, larger request have longer diameter. As the resilience increases, the diameter of the placement also increases. This is because the request needs to be placed in multiple datacenters, which increases the diameter. For resiliency of two and above, the diameter increases as the number of datacenter decreases. This is in contrast to the case of resiliency 1 as observed in the previous experiments. For resiliency 2 and above, the request needs to be placed in multiple datacenters. As number of datacenters in the cloud decreases, the average distance between neighboring datacenters increases. Hence, the distance between the datacenters in a placement increases.

A. VM partitioning

Now we study the performance of our heuristic algorithm to assign the VMs to the selected datacenters. Given the communication requirements between the VMs and the available capacity in each of the datacenters, the algorithm assigns VMs to the datacenters to minimize the inter-datacenter traffic. We compare the performance of the heuristic algorithm with two other algorithms: Greedy and Random. *Random* algorithm assigns random datacenter to each of the VMs. *Greedy* algorithm selects datacenters in the decreasing order of available capacity, and assigns as many VMs as possible. While selecting the VMs, it chooses VMs with maximum total traffic first.

In our experiment, we assign a 100 VMs request to the datacenters. The bandwidth required between these VMs was taken randomly between 0 and 1 Mbps. We studied the inter-datacenter traffic for assignment of these VMs to k datacenters, where $k = 2, \dots, 8$. The available resources in each of the datacenter were between $100/k$ and $200/k$. Hence, we were assigning 100 VMs to datacenters containing between 100 and 200 VMs. We ran the experiment 100 times and report the average.

Figure 10 plots inter-datacenter traffic as a function of number of datacenters for the three algorithm. For all the algorithms, inter-datacenter traffic increases as the number of datacenters increases. This is because, number of available VMs per datacenter decreases as the number of datacenter increases. As VMs are placed in more datacenters, inter-datacenter traffic increases. From the figure, we also see that greedy performs better than random by 10.2%, and heuristic algorithm performs better than greedy by 4.6%. Figure 11 shows the same plot where the datacenters did not have any excess capacity. Here, the inter-datacenter traffic for heuristic algorithm was 28.2% higher than the previous experiments, due to lower datacenter

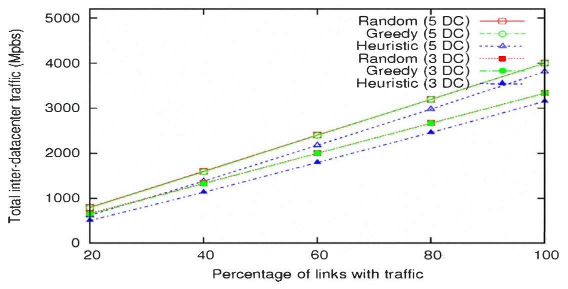

Fig. 12. Inter-datacenter traffic for VMs with partial traffic

capacity. Heuristic algorithm performed better than the other two algorithms by 4.8%. Performance of random and greedy was almost the same.

In figure 12, we study the effect of VM traffic on the inter-datacenter traffic. We vary the percentage of links with traffic in the VM communication graph from 20% to 100% and measure the inter-datacenter traffic. Here, the datacenter did not have excess capacity. We show the results for 3 and 5 datacenters. We observe that inter-datacenter traffic grows linearly with percentage of links with traffic for all the algorithms.

VII. Related work

Assigning individual VMs to datacenters and to CPUs within the datacenter falls under the category of graph partitioning problems. The problem is to divide the graph into pieces of equal size, such that there are few connections between the pieces. Even the simple case of dividing into 2 equal pieces, called graph bisection problem is NP-hard [4]. Well known Kernighan-Lin algorithm [11], and its variants [3], [10] are used for solving this problem heuristically. For partitioning graphs into several components, recursive bisections are used [2]. Most of these problems assume equal partition.

Another related problem is minimum $k - cut$ problem, which is to divide the graph into k sets, and minimize the total weight of edges whose ends fall in two set . If the sizes of the set are not given, but number of sets need to be r, then the problem is solvable in polynomial time [5]. If r is part of input, problem is NP-hard and $2 - 2/r$ approximation algorithm exists [15]. When the size of the sets are given and r is fixed, 3-approximation algorithm exists, if the weights follow triangle inequality [9]. Triangle inequality is not satisfied in VM assignment problem, as the weights are bandwidth required between the nodes.

A problem related to the datacenter selection is finding a maximum subgraph with a given diameter (MaxDBS) [1]. For general graphs, it is NP-hard to approximate MaxDBS to within a factor of $n^{1/2-\epsilon}$ for any $\epsilon > 0$.

The problem of assigning VMs inside a date center to minimize overall network costs has been studied in [13]. Here the objective is to place VMs that have large communication requirements, close to each other so as to reduce network capacity needs in the data center. A quadratic-assignment formulation of the traffic-aware placement problem is presented and solved with an approximation algorithm.

VIII. Conclusions

The main contribution of this paper is the development of algorithms for network-aware allocation of virtual machines in distributed cloud systems. In distributed cloud systems, inter-data-center latencies may be large and affect application performance when the VMs for an application are split over multiple data centers. Since the deployed cloud resources in each data center in a distributed cloud system are likely to be much smaller than in centralized data centers it is also more likely that a user request for resources gets split amongst multiple data centers. Hence, the use of good resource allocation algorithms is critical to achieving good application performance in distributed cloud systems. Based on the observation that VMs which are tardy in their completion times, due to communication latencies, can increase overall completion times for user requests (and so affect application performance), we developed data-center selection algorithms for VM placement that minimize the maximum distance between the selected data centers. We showed that this problem is NP-hard and developed a 2-approximation algorithm. The same algorithm can also be used for locating VMs inside data centers. However, inside data centers the network topology is often hierarchical and for this case we developed an optimal algorithm. We also developed heuristic algorithms that perform well for assigning VMs to processing resources in the chosen data center locations.

References

[1] Y. Asahiro, E. Miyano, and K. Samizo. Approximating maximum diameter-bounded subgraphs. *LATIN'10*, pages 615–626, 2010.
[2] C. E. Ferreira, A. Martin, C. de Souza, R. Weismantel, and L. Wolsey. The node capacitated graph partitioning problem: A computational study. *Mathematical Programming*, pages 229–256, 1998.
[3] C. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. *Design Automation Conference*, 1982.
[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.
[5] O. Goldschmidt and D. Hochbaum. Polynomial algorithm for the k-cut problem. *IEEE Symp. on Foundations of Comput. Sci.*, 1988.
[6] A. Gottlieb. Beware the network cost gotchas of cloud computing. *Cloud Computing Journal*, June 2011.
[7] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. *ACM SIGCOMM*, 2009.
[8] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM*, 2009.
[9] N. Guttmann-Beck and R. Hassin. Approximation algorithms for minimum k-cut. *Algorithmica*, page 198207, 1999.
[10] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Siam J. on Scientific Computing*, 1999.
[11] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 1970.
[12] T. Leighton. Improving performance on the internet. *Commun. ACM*, 52, February 2009.
[13] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. *INFOCOM*, 2010.
[14] 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*, 2009.
[15] H. Saran and V. Vazirani. Finding k-cuts within twice the optimal. *IEEE Symp. on Foundations of Comput. Sci.*, 1991.
[16] SCOPE Alliance. Telecom grade cloud computing. *www.scope-alliance.org*, 2011.