Two-Tier Multi-Tenancy Scaling and Load Balancing

Wei-Tek Tsai*, Xin Sun, Qihong Shao, Guanqiu Qi

Abstract

Cloud computing often uses the multi-tenancy architecture where tenants share system software. To support dynamically increasing demands from multi-tenants, the cloud service providers have to duplicate computing resources to cope with the fluctuation of requests from tenants. This is currently handled by virtualization and duplication at the application level in the existing cloud environment, such as Google App Engine. However, duplicating at the application level only may result in significant resource waste as the entire application is duplicated. This paper proposes a two-tier SaaS scaling and scheduling architecture that works at both service and application levels to save resources, and the key idea is to increase the resources to those bottleneck components only. Several duplication strategies are proposed, including lazy duplication and pro-active duplication to achieve better system performance. Additionally, a resource allocation algorithm is proposed in a clustered cloud environment. The experiment results showed that the proposed algorithms can achieve a better resource utilization rate.

1. Introduction

Cloud Computing has emerged as a new infrastructure that enables rapid delivery of computing resources as a utility in a dynamically scalable, virtualized manner. Typically, SaaS applications allow multiple tenants to reside on single or multiple instances of the software at the same time, which helps bring down the licensing fee. Virtualization is widely used in current cloud computing systems [1][2], which allows the ability to run multiple systems on a single physical system or one operating system on multiple physical systems as shown in Figure 1.

![Figure 1 Virtualization in Cloud](image)

To cope with dynamically increasing demands from multiple tenants, cloud service providers need to allocate computing resources for the application dynamically. Existing solutions often involve application instances duplication. For example, for a Google App Engine (GAE) application, as the traffic to the application increases, more instances of the application are duplicated and deployed on more servers for load balancing. However, most applications are componentized as software services, and computing load might not be distributed evenly among application components. Therefore, duplicating at the application level only may result in computing resource waste, as it is usually only the bottleneck component that needs more computing resource.

Another challenge is that different server nodes (virtualized or not) can have diverse levels of computing power. Given a resource request, how to choose the most suitable server nodes to run application/service duplicates leaves some questions.

This paper proposes a two-tier SaaS scaling and scheduling architecture that duplicates at both service and application levels along with a resource allocation algorithm that takes different computing power of server nodes into consideration. The main contributions of this paper are as follows:

- A two-tier SaaS scaling and scheduling architecture at both service and application levels;
- A resource allocation algorithm that selects suitable server nodes to run application/service duplicates.
- Two duplication time strategies, lazy and pro-active are provided to be chosen according to application requirements and
- Experiment evaluation that shows the effectiveness of the proposed algorithm.

The paper is organized as the following: Section 2 discusses server level and application level duplication and proposes a two-tier SaaS scaling and scheduling architecture; in section 3, a clustered-based resource allocation algorithm is proposed; section 4 discusses two different duplication time strategies and how to choose between them; section 5 shows some simulation results to show the effectiveness of the resource allocation algorithm; section 6 concludes the paper.

2. Two-Tier SaaS Scaling and Scheduling Architecture

To show why duplication at service level is necessary, several definitions are introduced first.
Application Request (R) is a request sent by the end users to the application. For a service-oriented application, several intermediate requests to the composing services might be generated. In Figure 2, intermediate requests, \(R_A, R_B, R_C\) is generated for service A, B, C for application \(R_{APP}\).

Component Throughput (Thr) is the maximal number of requests a component can process (in a second) without violating the SLA.

Overloaded/Underloaded Component: If the requests sent to the component are greater than the throughput, the component is overloaded, otherwise it is underloaded. The overloaded components can affect the system’s performance and result in SLA violation, while the underloaded ones may bring down resource utilization rate.

For example, in Figure 2, the number in the brackets indicates throughput of each component. The SLA for this application requires that a request to the component needs to be returned within 2 seconds. At a certain snapshot, suppose 15 requests are sent the application per second. Considering the following two cases of service requests of B and C:

\(<Case 1>\): Twice as many intermediate requests are generated for B as C: \(R_B = 2\ R_C\). After A, 10 \(R_B\) are sent to B, and 5 \(R_C\) are sent to C, therefore, the application can return these requests within 2 seconds successfully. The application is at the optimal condition, since all of its components are working at the maximal capacity.

\(<Case 2>\): if twice as many intermediate request are generated for C as B: \(R_C = 2\ R_B\). After A, 10 \(R_C\) are sent to C, 5 \(R_B\) are sent to B. All the requests to B can be processed in one second. Because C’s throughput is 5, it takes 2 seconds to process these 10 intermediate requests, or it can only process 5 of them in 1 second. Therefore, only 10 \(R_{APP}\) are successfully returned in 2 seconds. In this situation, B is underloaded, and C is overloaded.

For case 2, with GAE, to return all the 15 application requests within 2 seconds, the system needs to create another instance of the application, which leads to resource waste. C is the bottleneck of the application and needs to be duplicated. On the other hand, the underloaded services can be relocated to less powerful sever nodes to make better resource utilization.

2.1. Duplication Strategies

The duplication can happen in different granularities:

**Duplication Applications:** a similar strategy used by GAE, the coarsest granularity. Since applications are in service-oriented manner, both the application instance and the service instances belonging to it are duplicated as shown in Figure 3(a).

**Duplication Services:** New service instances will be created and deployed to servers if all existing instances of this service are overloaded as illustrated in Figure 3(b). With the increase of traffic, too many service instances will be duplicated under one application instance, which will eventually overload the service balancers.

**Duplication at a mixture of Service/Application:** If the duplication can happen at both application and service levels, shown in Figure 3(c), each load balancer might have fewer instances to manage, thus the balancing workload is further distributed on the application level.

2.2. Two-Tier SaaS Scaling and Scheduling Architecture

Based on the above analysis, this paper proposes a two-tier SaaS scaling and scheduling architecture described in Figure 3. Several components are as follows:

**Application/Service Container:** a runtime environment for an application/service which includes monitoring, resource management, security, and other supporting features.
**Re-deployable Service Package**: a package that contains the source code/compiled code of a service and related resources required to deploy the service on to a service container.

Containers running on newly provisioned server nodes will create additional service instances on service instances become overloaded, the load balancer routes requests to the different instances. When all the resource requests, the load balancer. According to information for each tenant in multi-tenancy architecture contain the customization service belonged to an application instance.

**Service Replica/Instance**: a concrete deployment of re-deployable service package on a service container, which is able to handle service requests. A service replica is also called service instance. A service replica can be customized to meet tenants’ specific requirements according to the tenant configuration files.

**Monitoring Service**: monitors the performance of a service/application instance. Based on historical data, it can report whether a service instance on its node is underloaded, overloaded or at its optimal condition.

**Service Load Balancer**: manages all instances of a service belonged to an application instance.

**Tenant Configuration Files**: contain the customization information for each tenant in multi-tenancy architecture that the load balancer can use when creating new application/service instances.

All the requests to a certain type of service are first routed to the load balancer. According to information from monitoring services, including each service instance’s throughput, status and configuration combined with the characteristics of requests, the load balancer routes requests to the different instances. When all the service instances become overloaded, the load balancer will create additional service instances on service containers running on newly provisioned server nodes using re-deployable service package.

### 3. Cluster-Based Resource Allocation Algorithm

Different server nodes (virtualized or not) can have diverse levels of computing power. When allocating server nodes to run application/service duplicates, the difference of computing power of servers should be taken into consideration.

One can cluster server nodes into different categories according their computing powers. Virtualization techniques can partition or merge multiple machines computing power to allow finer-grained resource allocation. Generally, the computing power of a server node/computer is determined by many factors, such as CPU speed, bus speed, cache size, RAM size and type, etc. When considering virtualization, certain overheads should also be considered, such as virtual network connectivity, scheduling, etc. How to model a server node’s computing power is not the focus of this paper and we assume that computation power of each server node can be modeled as a digit number. All the following sections will use the digit numbers to simplify the discussion. (More details of modeling computers’ power can be found in [4] [5].)

#### 3.1. Clustering Servers using Computation Power Model

Suppose N server nodes exist in a cloud. $P_j$ is the computing power number for server node $S_j$. $P_{\text{min}}$ is the lowest computing power number of all the nodes. We cluster the computers into different groups according to the following:

$$C_{2^i} = \{ S_j \mid P_j \in [2^i \cdot P_{\text{min}}, 2^{i+1} \cdot P_{\text{min}}), i \in N \}$$

For example, if $S_j \in C_1$, $P_{\text{min}} \leq P_j \leq 2P_{\text{min}} \cdot |C_{2^1}|$ is the total number of server nodes in cluster $i$.

It is a simple abstraction to build server nodes whose computing power increases in the power of 2. The computing power for a server node infinitely in reality, we define the $i$ deployment where $l$ is the highest $i$ value.

Two pools of services are maintained for each cluster $C_{2^i}$: $C_{2^i-\text{allocated}}$ contains all the server nodes that have been allocated for an application/service; $C_{2^i-\text{free}}$ contains the server nodes free to allocate.

#### 3.2. Cluster-Based Duplication Algorithm

The minimum allocatable resources are the server nodes belonging to cluster $C_1$, defined as $r_1$. Accordingly, a server node in $C_1$ represents 4 resource units, as $r_1$. In most of the cases, the request for computing resource is proportional to the how many requests the applications/services serve per second. We use $r_s$ to present a request resource that needs $X$ resource unit, then we have $s = r_s = r_s + r_s$. Let $t_{2^i}$ be the number of server nodes chosen from cluster $C_{2^i}$, a set $T = \{ t_{2^i} \mid t_{2^i} \in N, i \in N \}$ represents one way to allocate the resource.
Given a resource request \( r \), \( t_{total} = \sum_{i=0}^{l} t_{2i} \) is the total number of allocated server nodes of \( T \). \( t_{total} = \sum_{i=0}^{l} t_{2i} \) is sum resources of all the selected nodes of \( T \). \( T \) is the set that contains all \( T \).

### 3.2.1. Optimization Goal of Resource Allocation

The motivation is based on the following two considerations:

- **Avoid resource waste.** For example, a resource request \( r_{31} \), and we have two possible resource allocations \( T_1 \) and \( T_2 \): \( T_1 \) uses \( r_{16} \) resource, and \( T_2 \) uses \( r_{16} \). We prefer \( T_1 \) over \( T_2 \) as \( T_1 \) uses less resources.

- **Avoid scheduling overheads.** Generally, the more number of server nodes, the more scheduling overheads. For example, for a resource request \( r_{16} \), we have two allocations \( T_1 \) and \( T_2 \) that both use \( r_{16} \) resources. \( T_1 \) only uses one server node from cluster \( C_{16} \), \( T_2 \) uses two server nodes from cluster \( C_8 \). We prefer \( T_1 \) over \( T_2 \) as \( T_1 \) uses fewer server nodes.

Given a resource request \( r_x \), one first searches solutions with the least \( r_{total} \) without going below \( r_x \). If there are multiple candidates, we choose the one that uses the fewest server nodes. Formally, \( T_{\text{candidate}} = \{ T \mid T \in T, \ r_{total} \geq r_x \} \) as all the possible allocations that satisfy the resource request. \( r_{min} = \min \{ r_{total} \mid T \in T_{\text{candidate}} \} \) and \( T_{min-R} = \{ T \mid T \in T_{\text{candidate}}, \ r_{total} = r_{min} \} \).

The optimal allocation can be defined as follows:

\[
\text{Given a resource request } r_x, \ \text{find a set } T_{opt} \in T_{min-R}, \ r_{opt} = \min \{ r_{total} \mid T \in T_{min-R} \}
\]

### 3.2.2. Resource Allocation Algorithm

The algorithm uses a greedy approach to achieve the optimization goal as shown in Algorithm 1. Basically, the algorithm tries to allocate server nodes with computing power no greater than \( r_x \) in a descending order, meaning it allocates more powerful server nodes first till the server nodes in \( C_x \).

If the total resource of \( T \) is equal to \( r_x \), then it is the optimal allocation. However, some clusters might not have enough server nodes available, so it is possible that the total resource of \( T \) is less than \( r_x \). In this case, the algorithm retrieves back to the last level where adding one more server node would cause \( r_{total} \) to be greater than \( r_x \). It releases all the server nodes from lower levels, and adds one server node from this level. At this time, the \( r_{total} \) is the \( r_{min} \). The algorithm then releases all the server nodes (it is not really releasing since the system only allocates the resource after the optimal allocation is completed), and uses \( r_{min} \) instead of \( r_x \) to another round of allocation from the beginning and returns the allocation it finds.

**Algorithm 1: Resource Allocation (RA)**

**Input:** \( l \) level cloud clusters, given resource \( r_x \),

**Output:** An optimal allocation \( T_{opt} \)

**Signature:** \( T \ allocate(int r_x) \)

```c
for(int i = 0; i <= l; i++) //initialization
    t_{2i} = 0;
int j = l + 1;
// j is the last level adding one more server node of which // will cause \( r_{total} > r_x \)
for(int i = l; i >= 0; i--) {
    while (\(|C_{2i}| = 0 \& \& r_{total} + r_{2i} < r_x\) { 
        t_{2i}++; \(|C_{2i}|--; \)
    }
}
if(|C_{2i}| = 0) j = i;
if(r_{total} == r_x) return T;
for(int i = 0; i < j; i++) {
    t_{2i}--; \(|C_{2i}|++; \)
}
if(j == l + 1) return null;
else {
    t_{2j}++; \(|C_{2j}|--; \) int r_{min} = r_{total};
    release(T); //release server nodes in current allocation
    return allocate(r_{min});
}
```

**Example 1:** Considering a level 5 cluster \( C = \{ C_1, C_2, C_4, C_8, C_{16}, C_{32} \} \) given a resource request \( r_{31} \), find the optimal allocation \( T_{opt} \) using algorithm 1:

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
<th>Step 5</th>
<th>Step 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C_2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>C_4</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>C_8</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C_{16}</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C_{32}</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 1 Running Example for RA**

In Table 1, each value represents the current available resources in cluster \( C_x \) (\( x = 1, 2, 4..., 32 \)) at certain step (column). When getting the request of \( r_{31} \), the algorithm skips \( C_{32} \) as \( r_{32} > r_{31} \). It checks if there is any free server nodes in cluster \( C_{16} \) and allocate one if there is. After this step, the values are shown in column “step 1”. Specially, we circle \( C_{16} \) as one server node is allocated in this cluster. Allocating another server node from \( C_{16} \) will cause \( r_{total} \) to be greater than \( r_{31} \), therefore the algorithm allocates a server node from \( C_8 \) instead. Similarly, we keep on allocating server nodes till \( C_1 \). At this time, \( r_{total} = r_{30} < r_{31} \), and the last level in which adding one more server node will cause \( r_{total} \) to be greater than \( r_x \) is level 1, \( C_2 \). The algorithm adds one server node from \( C_2 \) to the allocation. Now, we have \( r_{total} = r_{32} = r_{min} \). The algorithm then releases all the server nodes in the allocation, and uses \( r_{32} \) to return the algorithm. It means that there is no allocation that just meets the resource request \( r_{31} \). In the second run using \( r_{32} \) as input, the algorithm allocates 1 server node from \( C_{12} \). Note there are other possible allocations that also have \( r_{32} \) resource, however, our
algorithm only returns the one with the fewest server nodes.

3.2.3. Resource Reallocation
When the usage of an application/service increases/decreases, resource might be added/withdrawn. However, the additional resource might cause the allocation not to be optimal. For example, initially a resource request for an application/service is \( r_6 \), an optimal allocation is to choose one server node from \( C_2 \), and one server node from \( C_1 \). When the usage increases, \( r_2 \) additional resource is needed. One possible solution is to add another server node from \( C_2 \); however, since the total resource needed now is \( r_6 \), we could deploy an application/service instance onto a server node from \( C_3 \), and retract the two nodes allocated previously. The algorithm for resource reallocation is shown as follows. 

<table>
<thead>
<tr>
<th>Algorithm 2: Resource Re-allocation (RR)</th>
</tr>
</thead>
</table>

**Input:** \( T \) level cloud cluster development, an existing allocation \( T \) whose \( r_{total}=r_x \), an additional resource \( r_y \).

**Output:** An optimal allocation \( T_{opt} \).

**Signature:** \( T \) re-allocate\( (T, \text{int } r_y) \).

\[
T' = \text{allocate}(r_{x+y})
\]

Compare the new allocation \( T' \) with the old allocation \( T \).

Create application/service instances on server nodes that were not in \( T \), and retract the server nodes in \( T \) that are not in \( T' \) now.

4. Lazy or Pro-Active?

All the discussions in Section 3 use a lazy way to handle service requests, which is “wait until needed”. While in some cases, when one can predict the demand request in the near future, pre-allocate resources can improve the system performance greatly. For example, the usage of tax return services usually bursts from January to April 15 every year; in that case, one can duplicate the service earlier and pre-allocate the resources needed, which in turn can improve the tenant’s performance.

To predict the future demand of services, multivariate exploratory techniques in data mining and machine learning can be applied to identify patterns for future service demand, such as Time Series Analysis.

Customers can choose lazy and pro-active models according to specific application requirement and QoS (Quality of Services): low penalty task and high penalty task. The first type means even the service requirement could not be satisfied, the penalty is low and acceptable, one can use lazy model and duplicate until must do so. While the latter has a high penalty when short of service supply, it is highly recommended to use pro-active model to plan and duplicate ahead and reduce the penalty. The model proposed in this paper provides flexibility for model choices.

5. Simulation Results

We compare our resource allocation algorithm with two straightforward algorithms.

- **Small First (SF) Algorithm:** allocate less powerful server nodes from lower level clusters first.
- **Big First (BF) Algorithm:** to allocate more powerful server nodes from higher level clusters.

**Simulation Setup:** We tested 3 three allocation algorithms with the following 5 level cloud deployments. \([C_1=10 \quad C_2=20 \quad C_3=30 \quad C_4=50 \quad C_5=80 \quad C_6=100]\)

We first randomly generated 200 resource requests between \( r_1 \) and \( r_2.0 \). Three parameters are evaluated: utilization rate (UR), server node number (SNN) and Success Rate (SR).

The utilization rate (UR) is defined as the ratio of total resource requested to total resource allocated, which indicates how effective computing resource is utilized. Figure 4(a) shows how UR changes as the request number increases to 100 for the three algorithms. UR (RA) decreases slowly as the request number increases and the lowest is around 0.94. The UR (SF) starts off strong then decreases sharply and the lowest comes at 0.72, because SF allocates server nodes from lower level clusters first, and as these server nodes quickly run out, it has no choices but to keep giving out more powerful server nodes even for smaller resource requests. UR (BF) first allocates server nodes from higher level clusters that possess more computing power than requested, which results in low UR in the beginning. As these server nodes with higher computing power number run out, BF starts to return allocations with allocated resource closer to the requested resource. The total allocated server node number (SNN) affects the performance of load balancers, the more service nodes the heavier workloads. Figure 4(b) shows how SNN for three algorithms. At the beginning, SNN for SF grows the fastest, followed by RA, then BF, since SF first allocates less powerful server nodes thus uses more of them for the same resource requests than other two. BF increasing rate grows higher, and at around 130th request, it surpasses the SNN of RA. Since BF aggressively allocates more powerful server nodes first, which leads to smaller SNN.

Success Rate (SR) measures the percentage of the resource requests allocated successfully within the 200 resource requests generated as shown in Figure 4(c). RA has the highest SR, And the numbers shown in the curly brackets are the differences of SF, BF compared to SR.
6. Related work

Ricardo and etc [6] first identified the importance of replication in enhancing multi-tier information system’s availability and scalability, then discussed several replication patterns. Huo and etc [8] proposed a Dynamic Service Replica Process (DSRP) system which creates and deletes service automatically to achieve better load balancing and performance. Various replication strategies, such as ‘active’ and ‘passive’ techniques, are presented by Maamar et al. [9]. Osrael et al. [10] proposed a generalized architecture for a service replication middleware. Juszczyk et al. [11] describe a modular replication architecture. Salas et al. [12], Engelmann et al. [13], and Laranjeiro and Vieira [14] propose an active replication framework for Web Services, a virtual communication layer for transparent service replication, and a mechanism for specifying fault tolerant compositions of web services using diverse redundant services, respectively. Tang and etc [15] discussed the load replication and balance in data centers. All these studies are sitting at the application levels, and try to describe various strategies for invoking and maintaining replicated applications. In this paper, we propose a new architecture, a mixed duplication policy at both application and service level, which can improve the resource utilization as well as system scalability.

7. Conclusion

This paper proposes a two-tier SaaS scaling and scheduling architecture, in which duplication can happen at both service and application levels to avoid resource wastes. A cluster-based resource allocation algorithm was proposed. Two duplication timing models, lazy and pro-active are proposed to work together with duplication. Simulation results showed that the resource allocation algorithm can provide better resource utilization rate while using fewer server nodes.

Acknowledgment: This project is sponsored by U.S. Department of Education FIPSE project P116B060433, U.S. National Science Foundation project DUE 0942453, and Department of Defense, Joint Interoperability Test Command. The research is also partly sponsored by the European Regional Development Fund and the Government of Romania under the grant no. 181 of 18.06.2010.

8. References