Resource State Monitoring of Service Transactions in Cloud Systems

Lei Xu*, Li Zhang†, Ewnetu Bayuh Lakew‡, Claus Pahl*
*Irish Centre for Cloud Computing and Commerce (IC4), School of Computing, Dublin City University, Ireland
{lxu,cpahl}@computing.dcu.ie
†Northeastern University, Shenyang, China
{zhangl}@swc.neu.edu.cn
‡Department of Computing Science, Umeå University, Sweden
{ewnetu}@cs.umu.se

***Abstract*—In cloud systems, services constituting a transaction may spread over a large number of servers or clusters. Theoretically, these services could consume cloud resources unlimitedly. To avoid financial loss due to resource overuse, clouds have to monitor the state of resources consumed by the services – collect values of consumption, and evaluate whether the combined usage of resources has excessed a pre-defined upper bound or not. The distributed nature of the services introduces a challenge to the monitoring system on how to summarise distributed state information with low cost. We present our resource state monitoring solution to capture the challenge introduced by services hosted in clouds. Our solution tracks the resource consumed by each service constituting a transaction individually whilst ensures the whole transaction does not overuse the allocated resource. It improves availability by avoiding single points of failure, and achieves scalability by minimising message exchanges. We performed experimental analyses that indicate this work can provide an inexpensive resource monitoring solution for transactions in clouds.**

I. Introduction

A transaction is a unit of work involving one or more services [1]. These services can be offered by multiple partners. Traditionally, transactions have the ACID properties: Atomicity, Consistency, Isolation and Durability. Either all services constituting these transactions are completed successfully to accomplish an agreed upon business goal or none of them have effect. Traditional transactions can be short lived and involve partners who are willing to cooperate under a single transaction manager. Contrarily, in large distributed systems such as clouds, services constituting transactions may be widely distributed and hosted in different clusters/zones. These transactions can be long lived – running for hours, days, or more, and may employ multiple coordinators due to various reasons such as high overhead on the communication between clusters/zones, or lack of trust. The ACID properties are not mandatory for long lived transactions in clouds.

In many cases, resources consumed by services constituting transactions are monitored, and a state alert will be triggered whenever a predefined threshold is violated. For example, a credit budget is set to control the cost of a transaction. This budget should be monitored in real-time over all services of the transaction to avoid cost overrun. Such resource state monitoring is essential for the observation, analysis and control of transactions. It can be achieved in the transactions that are short lived and only have one single coordinator by deploying a monitor component on the coordinator. This centralised solution simplifies the implementation and guarantees optimal utilization of resource.

Transactions in clouds may consist of services offered by multiple providers or one provider's different data centres. These services can be long lived and widely distributed. Theoretically, they are able to consume geographically dispersed resources unlimitedly. This could be risky since the cost of vast amounts of resources consumed unintentionally could be tremendous and may even exceed the estimated profit of the services [2]. To manage the services and limit global resource consumption, the providers may need to deploy multiple coordinators, and monitor the state of resource consumption.

For example, Dropbox is a storage cloud that keeps customers' data on Amazon Simple Storage Service (S3). Files of a customer may be stored in different clusters of Amazon's multiple data centres located across the United States. Based on the price plan, there is an upper bound on the storage space that can be consumed by a customer. However, this resource limit can not be allowed to each cluster of those data centres since demands at every cluster can not be known in advance. It should be supervised globally by multiple coordinators hosted in the clusters offering storage space to prevent resource overuse and revenue loss.

To manage resource consumed by transactions in clouds, the centralised monitoring approach is not applicable due to the distributed nature of the services constituting the transactions, and the involvement of multiple coordinators. The centralised management approach on long lived services can introduce high overheads in clouds due to the large amount of message exchanges [3], [4]. Meanwhile, when a lot of state information is received from services, the central component can become a bottleneck, and even worse, it is liable to single point of failure. Furthermore, the monitor component deployed on one coordinator may not be able to track resources controlled by another coordinator efficiently due to network delay.

One potential solution to tackle the issue presented above is to deploy a light-weight monitoring component on each service/coordinator of a transaction in a cloud to supervise

resource consumption and trigger state alerts. In such a case, how to allocate the resource budget to services/coordinators has to be investigated. Each service/coordinator may receive the overall budget. However, the services constituting a transaction can be invoked in parallel that may lead to resource overuse; resource budget may be allocated statically to each service/coordinator, i.e. every service/coordinator receives a fixed share. However, resources consumed by services can vary and may not be known in advance.

Our previous work [5] manages cloud resources between clusters, which is a single-layer resource management solution. To tackle the challenge faced by monitoring, we extend it to adapt to transactions in clouds, in which resource budget is allocated between services constituting transactions and/or subtransactions. Initially, resource budget is distributed to services of a transaction and managed by their monitoring components coined as "probe" in this paper based on the execution mode[1]. When the resource balance of a service is very close to or lower than its threshold, it will apply for resource re-allocation. The remaining resource budget is re-assigned to each service based its rate of consumption to avoid immediate global re-allocation and reduce communication overhead. We deploy such a re-allocation policy since the behaviours of consumers have inertia [6], i.e. the current resource consumption by one service depends on the past and influences future consumption. Once the overall resource budget is depleted, a state alert will be triggered.

This paper is structured as follows. Section II gives an overview of our solution including the transaction model, and our approach to initialise resource budget allocation and to re-allocate resource units. Section III specifies our allocation algorithms and the algorithm to reduce the overhead of re-allocation. Section IV describes the results of a simulation study undertaken to compare our solution with the one deploying a centralised coordinator for resource management. Section V provides a brief review of related work. Finally, Section VI summaries the contribution of the paper and outlines some future work.

II. Solution Overview

We investigate a solution to manage resources (e.g. storage, memory or network) consumed by services constituting a transaction. This solution is based on the transaction model in which services can be executed in parallel or sequentially. We assume that each service or coordinator hosts a probe to manage resources. The resource budget is allocated to services based on the manner to invoke them whilst it will be re-allocated once the balance of one service is lower than or close to its threshold. The budget will not be allocated to coordinators since they do not consume the resource directly by themselves. The probes of coordinators are only responsible for the coordination of resource budget re-allocation. The transaction model, the threshold to trigger re-allocation, and the way to calculate the number of resource units allocated to each service in re-allocations are presented as follows.

[1]Services constituting a transaction can be executed in parallel or sequentially in our work, which will be explained in Section II-A.

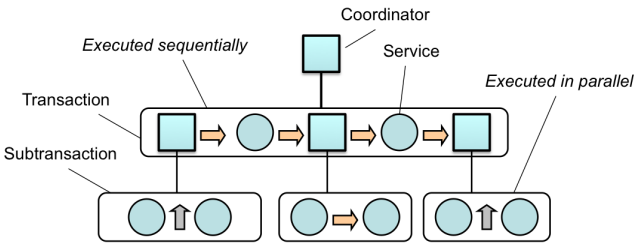

Figure 1: Nested Transaction.

A. Transaction Model

Transactions in clouds can invoke services of several different servers hosted in multiple clusters/zones. These transactions can be structured as flat or nested. In a flat transaction, a single coordinator is deployed to coordinate the execution of the services constituting the transaction, and these services are invoked sequentially or in parallel. In a nested transaction, as illustrated graphically in Figure 1, the top-level transaction opens subtransactions whilst these subtransactions can open further subtransactions down to any depth of nesting [7]. Similar to flat transaction, the services involved in nested transactions can be executed in parallel or sequentially. In practice, the services at the same level of a hierarchy may be deployed on different servers of one cluster or different clusters of one cloud zone.

In this paper, we only focus on nested transactions since flat transactions can be viewed as their subtransactions. Furthermore, we assume services/coordinators in nested transactions have levels of different priority, and ancestors always have higher priority than descendants.

B. Problem Statement

We formally state how the resource budget is initially allocated between multiple services, how the remaining budget is re-allocated when the balance of a service is very close to or lower than its threshold, and how a state alert is triggered whenever no enough resources can be further consumed. We let n_k denote an arbitrary node[2] of a transaction on which our probe is hosted. Let $n_k \cdot child$ and $n_k \cdot parent$ denote the child and parent of n_k respectively. Let n_{root} denote the outermost node of the transaction that is the root of the tree structure, where $n_{root} \cdot parent = \emptyset$. Meanwhile, if $n_k \cdot child = \emptyset$, then n_k is a leaf of the tree. Let y_b denote a set of nodes whose parent is n_b, where $\forall n_a \in y_b$, $n_a \cdot parent = n_b$, and $|y_b|$ denote the number of nodes constituting the set y_b.

We let R_b and r_k denote the allocated resource units allocated to children of n_b and to n_k respectively. Let $n_b \cdot child \cdot mode$ denote the execution mode of the children of n_b, whose value can be *sequential* or *parallel*, where $|y_b| > 1$.

When $n_k \cdot child = \emptyset$, we let $\Delta t_c(k)$ and $\Delta t_d(k)$ denote the time interval spent to calculate and distribute remaining

[2]A node can be a service or coordinator in our work.

resource units, and the time interval consumed to send allocation requests and receive replies respectively. In addition, let $\Delta t_h(k)$ denote the time interval that the probe on n_k spent on tracking resource consumption. In this work, we assume $\Delta t_c(k)$, $\Delta t_d(k)$, and $\Delta t_h(k)$ are known in advance. When the resource balance of n_k is lower than or very close to its threshold β_k that is calculated based on a time interval denoted as θ_k, it will trigger resource budget re-allocation. The θ_k can be calculated as:

$$\theta_k = \Delta t_c\,(k) + \Delta t_d\,(k) + \Delta t_h\,(k) \tag{1}$$

Let t_s and t_e denote the start and end of a time interval within which the resource on service node n_k is consumed continuously, where $t_e = t_s + \theta_k$. Moreover, let $f(n_k, t)$ denote the consumption rate of the resource hosted in n_k, which may vary with consumption time denoted as t. Based on the above, β_k can be stated as:

$$\beta_k = \int_{t_s}^{t_e} f\,(n_k, t)\,dt \tag{2}$$

If n_h is a coordinator node and receives an allocation request, it will collect allocable resource units from its children. Otherwise, if n_h is a service node, when n_h receives a request from n_k[3], and decides to respond to it, its probe reserves a number of resource units that will be consumed within a given time interval $\Delta t_h^{'}$. This reservation is intended to avoid the suspension of service deliver due to low balance of the resource during re-allocation. Afterwards, n_h reports the number of resource units that can be re-allocated, the resource consumed since the last allocation to n_k.

We let $\Delta t_h^{'} = \theta_h + (\theta_k - \Delta t_d\,(k,h))$, where $\Delta t_d\,(k,h)$ denotes the time interval spent to forward the request from n_k to n_h. This reservation is intended to avoid the submission of a re-allocation request from n_h before n_k completes the allocation. We define the threshold for resource reservation on n_h as $\beta_h^{'}$, which can be calculated by applying Equation 2 once $\Delta t_h^{'}$ has been identified. When n_h receives the re-allocation request from n_k, if its resource balance denoted as $r_h^{'}$ is less than what will be consumed within $\Delta t_h^{'}$, then it has to keep all the resources and will trigger re-allocation before n_k sends extra resource units to it. We will discuss how to tackle the issue that a probe requests resource budget re-allocation whilst another probe has not completed its allocation in section III-B. If there are enough resource units remaining, then n_h will forward $r(h) = r_h^{'} - \beta_h^{'}$ to n_k, where $r(h)$ denotes the number resource units that will be sent to the requester.

If the probe of n_k is qualified to coordinate the re-allocation, it will calculate the total number of allocable resource units remaining in children of n_b denoted as $R_b^{'}$, where $n_k \cdot parent = n_b$. Given the above, if $|y_b| > 1$, then $R_b^{'}$ can be calculated as:

$$R_b^{'} = \sum_{j=1}^{|y_b|} r(h), where\ r(h) = \begin{cases} 0 & r_h^{'} - \beta_h^{'} \leq 0 \\ r_h^{'} - \beta_h^{'} & r_h^{'} - \beta_h^{'} > 0 \end{cases} \tag{3}$$

If enough no resource units remain in the children of n_b and $n_b \neq n_{root}$, then n_k forwards the allocation request to its parent. Otherwise, $R_b^{'}$ will be distributed to the nodes based on the resource consumption[4]. Let n_a denote an arbitrary child of n_b, where $n_a \cdot parent = n_b$, and let Δr_a denote the resource units consumed on n_a since last distribution. Based on the above, the resource units n_a will receive after the re-allocation, which are denoted as $r_a^{''}$, can be stated as:

$$r_a^{''} = R_b^{'} \times \frac{\Delta r_a}{\sum_{j=1}^{|y_b|} \Delta r_j} \tag{4}$$

If $n_a \cdot child \neq \emptyset$, then n_a will allocate $r_a^{''}$ to its children once n_k completes its re-allocation.

III. Resource Management Algorithm

The resource budget is initially allocated to leaf nodes of the tree structure and the remaining resource units will be re-allocated between these nodes when necessary. The approach to initialise the resource unit allocation, the algorithm to apply for re-allocation and to response to the requester are presented in this section.

A. Initialisation

The allocation starts with the root node and deploys the breadth-first strategy to assign resource units. In case a node owns multiple children, if these children are executed in parallel, then the resource units of the parent is allocated to its children equally; if the children are invoked sequentially, then all resource units of the parent are given to the child that will be the first one to consume the resource[5]. In case a node only has one child, all its resource unit will be assign to the child. It is a recursive process and it will continue until all resource units have been allocated to leafs (services of the hierarchy).

To investigate how the branching factor and depth of the hierarchy affect the initialisation of resource units, the scalability of the approach is analysed as follows. Let M denote the maximum branching factor of the hierarchy whilst let N denote the height of the hierarchy. If the execution mode of all nodes in the hierarchy is parallel, then the resource units are allocated to up to $\sum_{i=1}^{N} M^i$ nodes; if all nodes of the hierarchy are executed sequentially, then the resource units are allocated to N nodes. Given the above, the computational complexity of the initialisation approach varies between $O(N)$ and $O(M^N)$.

[3]The n_k can be a sibling of n_h, where $n_h, n_k \in y_b$ and $n_b \cdot child \cdot mode = parallel$. Otherwise, n_k is a service node triggering the re-allocation in the node hierarchy. In this case, the request from n_k will be forwarded by the parent of n_h to n_h whilst the reply of n_h will be sent to n_k via the parent of n_h.

[4]If a node is a service, then the re-allocation is calculated based on resource units consumed by itself. Otherwise, the re-allocation takes into account all the resource units consumed by its children.

[5]Once the service has been delivery, the remaining resource units will be forwards to other services in the sub-tree or to other sub-trees.

B. Requester Side Allocation

When the resource balance of a node n_k is very close to or lower than its threshold β_k, where $n_k \cdot child = \emptyset$, the resource re-allocation can be triggered. This is depicted in lines 2-4 of Algorithm 1.

If a node requests extra resource units and it is the root of the tree structure, then a state alert is triggered. Otherwise, the algorithm tries to re-allocate resource units as follows. When a node requests an allocation and it is the only child of its parent or the children of its parent are invoked sequentially, then the parent of the node may need to activate a re-allocation process. This is depicted in line 8-9 of Algorithm 1. Otherwise, the node has to check whether another allocation is being processed to avoid inconsistencies due to multiple nodes re-allocating resources in parallel before it starts to collect resource units from others. If there are no other nodes requesting resource re-allocation, the node will multicast its requests to all its siblings.

As we discussed above, a node can start its own re-allocation when there are no re-allocations requested by other nodes are being processed based on its knowledge. However, in practice, this consistency can be tricky to achieve due to network delay, i.e. a re-allocation request has been submitted by another node whereas the node does not receive the request when it applies for re-allocation. To tackle this issue, we assume the priority levels of nodes in the hierarchy are different[6]. A lower priority requester has to terminate its allocation request if an allocation triggered by a higher priority node is processing. This will be further discussed in Section III-C.

Upon receipt of allocation requests, a node only relies to the requester with higher priority. The requester then calculates the total number of allocable resource units by applying Equation 3. If there are enough units remained, where $R_p^{'} > \beta_k$ and $n_k \cdot parent = n_p$, the node n_k calculates the resource units that should be assigned to its siblings by applying Equation 4, and then forwards them to the nodes based on the calculation results. Otherwise, the parent of the requester needs to trigger a re-allocation process. The re-allocation for the node with multiple siblings that are executed in parallel is depicted in lines 11-23 of Algorithm 1.

Once the resource units have been re-allocated, all probes hosted in services will continue supervising resource consumption. This is a recursive process until the transaction is terminated or a state alert is triggered.

1) Complexity Analysis: As discussed in Section III-A, we let M denote the maximum branching factor of the node hierarchy whilst let N denote the height of the hierarchy. The most expensive case to process an allocation request is that an allocation is triggered by a leaf node whereas only children of the root have enough resource units remained. In that case, if no other allocation is processing, a requester needs to forward its request and receive replies from up to $M-1$ nodes at each level of the hierarchy. In full node hierarchy, requests and replies are sent $2N(M-1)$ times whilst the allocable resource units are allocated to up to $(M-1)+\sum_{i=2}^{N} M^i$ nodes. Based on above, the computational complexity of the algorithm is $O(M^N)$.

2) Improvement: The theoretical analysis indicates the computational complexity of the requester-side allocation algorithm is $O(M^N)$, which means the overhead of the algorithm can grow quickly with an increase of N[7]. To reduce the runtime complexity of the allocation solution, we could map the multiple level transactions into "flat" ones whose height is 1, and then apply the processed transactions as the input of our requester-side allocation algorithm. The mapping can be achieved by applying Algorithm 2. This algorithm is intended to divide all service nodes constituting the hierarchy into sub-groups in which services are executed in parallel, and then apply these sub-groups to Algorithm 1 sequentially. As a result of the mapping, the complexity of the requester-side allocation algorithm is reduced to $O(M)$.

In this algorithm, we let s denote a service node set that

Algorithm 1 Resource Allocation (Requester Side).

```
1:  loop
2:      if |r_k − β_k| < ε, n_k · child = ∅ then
3:          ALLOCATION(n_k, n_k · parent)
4:      end if
5:  end loop

6:  function ALLOCATION(n_k, n_p)
7:      if n_p ≠∅ then
8:          if n_p · child · mode = sequential or |y_p| = 1 then
9:              ALLOCATION(n_p, n_p · parent)
10:         else
11:             if no allocation has higher priority then
12:                 multicast request to all nodes in y_p
13:                 calculate R_p'
14:                 collect information on consumed resource
15:                 if R_p' > β_k then
16:                     allocate and distribute R_p'
17:                 else
18:                     ALLOCATION(n_p, n_p · parent)
19:                 end if
20:             else
21:                 terminate the allocation
22:                 reply to the node has higher priority
23:             end if
24:         end if
25:     else
26:         trigger state alert
27:     end if
28: end function
```

[6]We assume that descendants always have lower priority than ancestors. Based on that, the priority level of nodes can be set by various criteria that is outside the scope of this work.

[7]However, we would expect that in most cases the hierarchy of service transactions are flat, i.e. a single transaction constructed using transactions.

Algorithm 2 Pre-Process.

1: **while** $|S| > 0$ **do**
2: **for all** $n_k \in S$ **do**
3: **if** $n_k \cdot next \neq$ø **then**
4: add n_k to s, remove n_k from S
5: **if** n_k and $n_k \cdot next$ are siblings **then**
6: $n_p = n_k \cdot parent$
7: **else**
8: find out n_p, which is the parent of both n_k or its ancestor(s), and $n_k \cdot next$ or its ancestor(s),
9: **end if**
10: **if** $n_p \cdot child \cdot mode = sequential$ **then**
11: break
12: **end if**
13: **else**
14: add n_k to s, remove n_k from S
15: **end if**
16: **end for**

17: apply s as the input of the requester-side algorithm
18: remove all nodes from s

19: **if** state alert triggered **then**
20: terminate
21: **end if**
22: **end while**

will used as the input of our request-side allocation algorithm, and the parent of nodes included in s will be set as the root of the hierarchy. Let S denote another node set including all service nodes of a transaction, and let $|S|$ denote the number of service nodes constituting S. In addition, let $n_k \cdot next$ denote the node next to n_k in S.

In Algorithm 2, we assume service nodes in S are well sorted – a service node following another one in S is always on the right of that node in the hierarchy. Meanwhile, service nodes in S are associated with the information on the node hierarchy, including their siblings, ancestors and the execution modes.

The nodes of S are checked sequentially: when a node n_k is not the last one of the set S, it will be inserted into s. The checking process will continue in the following cases: when n_k and $n_k \cdot next$ are sibling, and their execution mode is not "sequential"; when the execution mode of the children of n_p is not "sequential", which is the parent of both n_k or its ancestor(s), and $n_k \cdot next$ or its ancestor(s). In addition, if a node is the last node of S, it will simply be inserted in s. This is depicted in lines 2-16 of Algorithm 2. The set s is then used as the input of the allocation algorithm. This will continue until all the services in S have been processed or a state alert is triggered.

The computational complexity of this mapping algorithm depends on the number of trees/sub-trees containing service nodes. Based on the notation given in Section III-A, the complexity is $O(M^{N-1})$. We believe this is acceptable since the algorithm will be typically be run off-line. Meanwhile, the number of services making up a transaction will be in the tens at the very most, and the number of trees/sub-trees containing the services will be less than that.

Algorithm 3 Resource Allocation (Responder Side).

1: **if** $n_{rec} =$ø **then**
2: calculate and forward resource units to n_{req}
3: send information on consumed resource units to n_{req}
4: $n_{rec} = n_{req}$
5: **else**
6: **if** $P(n_{req}) > P(n_{rec})$ **then**
7: retrieve r_{rec} from n_{rec}
8: calculate and forward resource units to n_{req}
9: send information on consumed resource units to n_{req}
10: $n_{rec} = n_{req}$
11: **else**
12: no response
13: **end if**
14: **end if**

C. Responder Side Allocation

We assume each node in the hierarchy keeps a record of the node to which it responded, and on the number of resource units sent to the recorded node until the re-allocation is completed. Let n_{rec} denote the recorded node and r_{rec} denote number of allocable resource units reported to the requester. Upon an allocation request, the node calculates its allocable resource units and sends the information to the requester if no other allocation is processing. This is depicted in lines 1-4 of Algorithm 3.

Let $P(n_{req})$ denote the priority level of the requester. When a node receives a request whilst it has replied to another node n_{rec}, it checks the priority level of the nodes. If n_{req} has higher priority, the node re-calculates its allocable resource units and forwards the information to n_{req}. In this case, when n_{rec} receives the request from n_{req}, it terminates its allocation and ignores all replies, and then responds to n_{req}. If n_{rec} has higher priority, the node will not respond to n_{req} whilst n_{req} has to reply to n_{rec}. This is depicted in lines 6-13 of Algorithm 3.

IV. Experimental Evaluation

In our experiments, we assume a transaction consists of multiple services. To maintain quality of service in the form of fast response time, we assume each of the services will consume of a unit of storage space for caching when they are serving a request on them. Although such storage space can vary when invoking different services in practice, we assume it is constant for all the services in our experiments for simplicity sake. Moreover, the consumed storage space can be released when services have delivered, but we only observe the amount

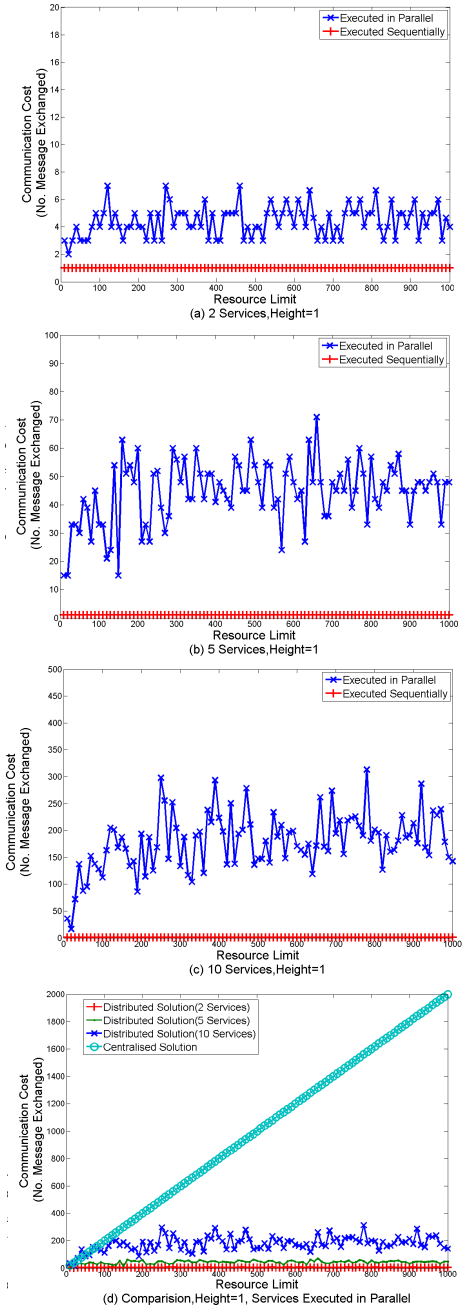

Figure 2: Communication cost with varying number of services.

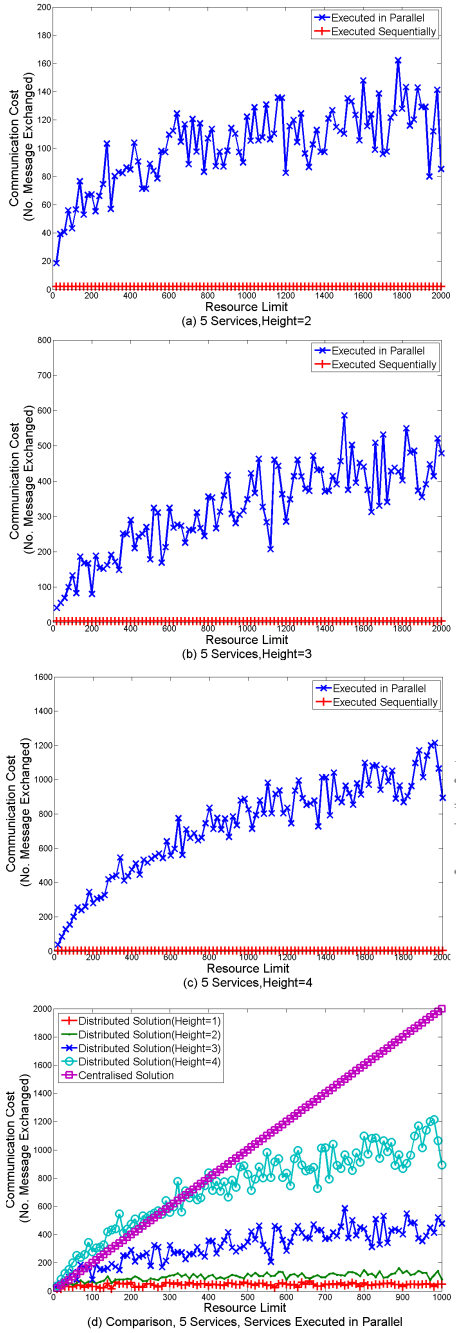

Figure 3: Communication cost with varying height.

of space been used for caching, not the space being occupied in real-time.

Services constituting a transaction can be invoked in different ways: if services are invoked sequentially, consumption requests will be forwarded to each of the services in sequence; if services are consumed in parallel, consumption requests will be forwarded to the services based on a Zipf distribution. We employ such a distribution since it has been conjectured that Zipf's law governs lots of features of the Internet [8]. This includes the distribution of requests for services.

The consumption requests on services constituting a transaction are generated by an generator. The request generator and the services are implemented in Java as RESTful web services. They are deployed on Tomcat servers hosted on our servers that are connected by our campus intranet. Different consumption limits are set in the experiments. Once the resource consumed by a transaction reaches the pre-defined limits, a state alert will be triggered.

We have analysed the upper bound of the communication overhead for a single leaf node to trigger and complete a re-allocation in Section III-B1. In this section, we try to investigate how the overall cost of the solution varies with the increase in height of transaction hierarchy and number of

service nodes constituting transactions. We change the height from 1 to 4 and the service nodes from 2 to 10 in the experiments. Meanwhile, we compare the number of message exchanged in the transactions whose height and number of service nodes are the same whereas services constituting them are executed in different manners. The communication cost we discussed in our experiments is measured in terms of the number of messages changed. Obviously, the configuration of our experiments are not able to give the full picture on the performance of the approach. However, we believe the results from our experiments indicate how the overhead of the solution varies with different transactions.

We first vary the number of services. The number of service nodes is set as 2, 5 and 10, the height of the node hierarchy is set to 1, and the total allocated resource units vary between 10 and 1000. As the results depicted graphically in Figure 2 (a)-(c), the cost of managing resource units consumed by services executed sequentially is lower than that of the services invoked in parallel. Meanwhile, Figure 2 (d) illustrates that the communication cost increases as more services are involved in the transactions in which all nodes are executed in parallel. Furthermore, even when all services are executed in parallel, we notice that the communication overhead of our solution is lower than that of the centralised approach in most circumstances. Note that in the centralised resource management approach, we assume a service will apply for resource from the centralised coordinator every time it receive a request. Upon the request of the service, the coordinator will grant a resource unit and forward its reply to the service.

Afterwards, we set the number of service nodes to 5, vary the height of the node hierarchy from 1 to 4, and change the total allocated resource units between 10 and 1000. As the results illustrated in Figure 3 (a)-(c), it is more expensive to manage resource consumed by services executed in parallel than those used by services invoked sequentially. Meanwhile, Figure 3 (d) indicates that the communication cost rises when the height of the node hierarchy increases, and the cost of our solution is lower than that of the management solution deploying a single centralised coordinator as more resource units consumed, even when all services constituting the transaction are invoked in parallel. We also notice that when the height of the transaction is 4, the overhead of our solution is higher than the centralised one in some test cases. In this case, we can apply Algorithm 2 to reduce the cost.

V. Related Work

Monitoring of services that form transactions is fundamental as monitoring data are the primary input for business analysts to track their business goals of transactions. The monitoring data are collected in an intra-organizational [9] [10] or a cross-organizational manner [11] [12]. They will be propagated to a centralised monitoring component in which the collected state information is evaluated against pre-defined monitoring criteria in near real-time. The monitoring systems are centralised and most of them are not applicable to clouds in which service transactions can spread over a large number of clusters/zones.

Compared with service monitoring in business transactions, resource management in distributed systems has been a more active research area for many years. One challenge that needs to be tackled in this area is how to manage and summarise resource consumption based on collected information from distributed nodes. A typical solution is to deploy a single coordinator to correlate the information, and process the data for various purposes including finding top-k [13], computing sums and counts [14] [15], and evaluating whether a pre-defined threshold is violated or not [3], [4], [16]. Again, resources are managed by a centralised coordinator in these works, whereas our solution is fully distributed and each probe can work as the coordinator.

Except for ensuring that a transaction does not overuse resources, our work presents a solution to re-allocate resource budget dynamically. Similar to our allocation solution, Raghavan et al. [17] present their approach to manage network bandwidth consumed by services distributed in a cloud. Services can continue consuming bandwidth unless the local granted resource is depleted. In the case, further consumption requests will be denied even though the granted bandwidth is not depleted globally. Karmon et al. [18] utilize a tree structure to distribute and enforce quotas in grids. Consumption requests will only be served when there is enough quota remaining in the node locally. Compared with the two solutions on resource allocation, our work is intended to make full use of the resource and tries to avoid the deny of consumption requests when resource units are depleted in a node, whereas enough resource remains in the cloud.

J. Behl et al. [2] present a distributed quota enforcement protocol. In this solution, free quotas are equally distributed between nodes running applications consumed by a given customer. Once the local balance of a node is changed, free quotas will be re-allocated. Compared with our solution, this work always tries to balance the free quotas of nodes no matter they consume quotas or not.

VI. Summary and Future Work

Service monitoring in business transactions and resource management in distributed systems have activated a diverse set of researchers to contribute their solutions to tackle challenges in the two areas. The proliferation of cloud services has led to an increasing demand to mange resources consumed by business transactions in large-scale distributed systems. To the best of our knowledge, our work can be the first state monitoring solution to track resources consumed by cloud services constituting transactions, and evaluate whether a pre-defined threshold is violated or not.

In our solution, each node – a service or a coordinator of cloud transactions is equipped with a probe for resource monitoring and re-allocation. In practice, this can be achieved by embedding our algorithms into the service level monitoring information providers, such as the "probe" of the Lattice monitoring framework [19].

Our re-allocation is processed between the node triggering the allocation and its siblings, or between the ancestors of

that node. The remaining resource budget is allocated to each service involved in the allocation based on its consumption rate to avoid further immediate re-allocations. When all the resource units allocated to a transaction is depleted, a state alert will be triggered. This resource state monitoring and management solution is fully distributed and there is no central coordinator, i.e. no central failure point involved. Compared with a centralised solution that deploys a single coordinator to control and manage resource consumptions, our work could increase performance, scalability, and fault tolerance.

Furthermore, we analysed the computational complexity of the our solution. As the height of a transaction hierarchy increases, the overhead of the solution could grow. To tackle this issue, we presented an algorithm to map multiple-level transactions into "flat" ones. It divides transactional services into different sub-groups. Multiple services included in each of the sub-groups are executed in parallel whilst the sub-groups are invoked sequentially. Additionally, we performed an experimental analysis to compare our solution with the centralised one. Our experimental results show that our solution can reduce the communication overhead in most circumstances.

We notice that our resource state monitoring solution lacks practical validation. Future work will initially concentrate on this. We will deploy our monitoring solution to track service transactions on Amazon EC2. It will further validate the solution, and complement the analyses and experiments presented in this paper.

Secondly, we assume the time consumed on communication and tracking of resource consumptions is known in advance in this work. This may not always be the case in practice, especially the time spent on message transmission. Currently, the time consumed on communication used in the test is calculated based on its upper bound and lower bound measured in the test bed whilst the time spent on tracking is obtained by estimation. It is common knowledge that there is always a minimum transmission time, which can be obtained if no other network traffic exist. However, most distributed systems are asynchronous in practice, and it can be hard to find out the upper bound on message transmission delays. We will investigate how to tackle this issue and relax the assumption in future.

Thirdly, we will investigate how to set the reservation threshold dynamically that can tolerate network dynamics and mitigate their impact on the monitoring system. Currently, the threshold is calculated based on the estimated time spent on communication and monitoring. As we know, network traffic changes constantly. The static threshold value can be too high or low. High threshold value triggers frequent re-allocations leading to high communication cost even when not necessary, whereas low threshold value may starve services when performing re-allocations.

Finally, it is not uncommon that services or probes fail in transactions. This has not been covered by this work. We will enhance our solution to enable nodes that dynamically replace failure services/coordinators or crashed probes to get knowledge on consumed resources and available balance. This will add complexity, however we believe it is required, given the prevalence of service/server, or network failure in clouds.

REFERENCES

[1] M. Verma and P. Deswal, "Approaching web services transactions," *IBM developerWorks Technical Library*, 2003. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-tranart/

[2] J. Behl, T. Distler, and R. Kapitza, "DQMP: A decentralized protocol to enforce global quotas in cloud environments," in *Proceedings of the 14th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 217–231.

[3] S. Meng, S. R. Kashyap, C. Venkatramani, and L. Liu, "REMO: Resource-aware application state monitoring for large-scale distributed systems," in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 248–255.

[4] S. Meng, L. Liu, and T. Wang, "State monitoring in cloud datacenters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1328–1344, 2011.

[5] E. B. Lakew, F. Hernández-Rodriguez, L. Xu, and E. Elmroth, "Management of distributed resource allocations in multi-cluster environments," in *the IEEE 31st International Performance Computing and Communications Conference, IPCCC'12*, Austin, TX, USA, 2012, pp. 275–284.

[6] M. Solomon, G. Bamossy, S. Askegaard, and M. Hogg, *Consumer Behaviour: A European Perspective (3rd Edition)*. Prentice Hall, 2006.

[7] G. F. Coulouris and J. Dollimore, *Distributed systems: concepts and design(5th Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2011.

[8] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet," *Glottometrics*, vol. 3, pp. 143–150, 2002.

[9] L. Baresi and S. Guinea, "Towards dynamic monitoring of WS-BPEL processes," in *ICSOC 2005, Third International Conference of Service-Oriented Computing, volume 3826 of Lecture Notes in Computer Science*. Springer, 2005, pp. 269–282.

[10] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, "Run-time monitoring of instances and classes of web service compositions," in *Proceedings of the IEEE International Conference on Web Services*, ser. ICWS '06, Washington, DC, USA, 2006, pp. 63–71.

[11] B. Wetzstein, D. Karastoyanova, O. Kopp, F. Leymann, and D. Zwink, "Cross-organizational process monitoring based on service choreographies," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 2485–2490.

[12] S. Wagner, C. Fehling, D. Karastoyanova, and D. Schumm, "State propagation-based monitoring of business transactions," *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications, SOCA'12*, vol. 0, pp. 1–8, 2012.

[13] B. Babcock and C. Olston, "Distributed top-k monitoring," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '03. ACM, 2003, pp. 28–39.

[14] R. Keralapura, G. Cormode, and J. Ramamirtham, "Communication-efficient distributed monitoring of thresholded counts," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '06. ACM, 2006, pp. 289–300.

[15] S. Kashyap, J. Ramamirtham, R. Rastogi, and P. Shukla, "Efficient constraint monitoring using adaptive thresholds," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ser. ICDE '08. IEEE Computer Society, 2008, pp. 526–535.

[16] S. Meng, S. R. Kashyap, C. Venkatramani, and L. Liu, "Resource-aware application state monitoring," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2315–2329, 2012.

[17] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud control with distributed rate limiting," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 337–348.

[18] K. Karmon, L. Liss, and A. Schuster, "GWiQ-P: an efficient decentralized grid-wide quota enforcement protocol," *Operating Systems Review*, vol. 42, no. 1, pp. 111–118, Jan. 2008.

[19] S. Clayman, A. Galis, and L. Mamatas, "Monitoring virtual networks with lattice," in *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, 2010, pp. 239–246.