Interacting with Service Workflows in a Cloud

Qianhui Liang

Singapore Management University
Singapore

althealiang1@acm.org

Michael N. Huhns

Department of Computer Science and Engineering
University of South Carolina USA
huhns@sc.edu

ABSTRACT.

When Web services are moved into a cloud computing environment, each service can be scaled over more servers to handle increases in demand. However, there is still the problem that external applications using the services must move control and data information in and out of the cloud. If there is a large amount of data, it would be more efficient for the data to flow directly among the services in the cloud, rather than to and from the external controlling application. Currently, there is no way to specify this using SOAP + WSDL, or REST. To achieve the full benefits of cloud computing, we describe in this paper a technique for transitional storage called cloud data containers to be placed within the cloud to hold intermediate process data, such that the data traffic to and from the cloud can be reduced substantially. The assignment of the physical data containers to individual services relies on service dependency relationships modeled by a directed graph that we term a *weighted service dependency graph*. Dependent services share one or more data containers. We have designed a data container selection scheme to identify the containers best able to transmit data from one partner service to another. The validity and efficiency of our data container assignment and selection scheme are demonstrated by experiments.

Keywords

Cloud data containers, data traffic, intermediate process data, transitional storage, service flow, weighted service dependency graph

1. INTRODUCTION

Service-oriented architectures (SOAs) have gained momentum recently as a way to improve the flexibility and reusability of software components [1][2]. Software components as services expose their interfaces publicly to allow message exchanges between each other via such interfaces and usually over a network. Messages carry the input and output data or error information of the internal execution of the services. Message transmission over the network and parameter encoding and decoding at both the service provider and consumer end points constitute a major external cost, which is not intrinsic to the processing of the service request by the service provider. This also reveals that scalability of service invocation has not been addressed by the paradigm of SOA itself. This is where cloud computing comes in to help with the scalability issue.

Qianhui Liang was with School of Information Systems, Singapore Management University, Singapore when this research was performed.

Cloud computing is an Internet-centric software model that features a scalable, multi-tenant, multi-platform, multi-network, and global software development model [9]. It encompasses a variety of aspects of software applications ranging from deployment, load balancing, provisioning, business model, and architecture. There are several views of clouds, three of which are SaaS, PaaS, and IaaS referring to Storage-, Platform-, and Infrastructure-as-a-Service, respectively.

Amazon provided the initial impetus for cloud computing when it announced its Elastic Compute Cloud (EC2) [5] and Simple Storage Service (S3) [4]. This happened because Amazon had many servers (more than 200,000) with a large amount of excess capacity. EC2 provides CPU cycles, and it is a good environment for executing an application that users access remotely. If the demand for the application becomes too large and its response time becomes too poor, then Amazon will just duplicate it to run on more servers. In particular, clouds can be seen as a new model of Internet-scale computing, whereby within minutes or even seconds they can provide a customer with just the right amount of computing power as demands fluctuate. Therefore, applications can exhibit linear or super-linear scalability.

What is gained by moving services into a cloud? The basic idea is that when services are moved into a "cloud," each service can be scaled over more or fewer servers to handle increases or decreases in demand. However, there is still the problem that control and data information must flow in and out of the cloud to any external applications using the service. If there is a large amount of data, it would be more efficient for the data to flow directly among the services in the cloud. Currently, there is no way to specify this using SOAP [7] + WSDL [8], or REST [9]. To achieve the benefits, we describe in this paper a technique for transitional storage called cloud data containers to be placed within a cloud to hold intermediate process data, such that the data traffic to and from the cloud can be reduced.

Elasticity and virtualization of resources in the cloud are enabled by the effective organization of the distributed resources and their supporting infrastructural auxiliaries (like the aforementioned data containers) in the cloud. Cloud applications must exchange data when using and sharing resources. When data have to be physically transmitted to provide virtualization, we need to guarantee such transmission is always minimized in the cloud. Data access patterns of cloud applications, to a large extent, determine how the resources and their auxiliaries must be deployed with servers in order to allow the most effective way of data transportation in the cloud.

We address the above need by capturing the data access patterns of cloud applications in the form of service dependency relationships. We believe data exchanges between different services (or, alternatively, between a service and a workflow engine) form the vast majority, if not all, of the data transmissions in the cloud. (Examples of other forms of data transmission can be messages for coordinating resource

allocation among different nodes in the cloud.) We analyze the service dependency relationships in order to calculate the best way of organizing the data containers with the servers. In particular, physical data containers are assigned to individual services. Such assignment relies on service dependency relationships, which we model in a weighted service dependency graph (WSDG). Dependent services share one or more data containers. We have designed a data container selection scheme to identify the containers best able to transmit data from one partner service to another. The validity and efficiency of our data container assignment and selection scheme are demonstrated by experiments.

The remainder of the paper is organized as follows: Section 2 is related work. Section 3 describes the motivation of this research using an example of composed services in a cloud and its interactions with an external application. Section 4 presents the idea of *cloud data containers*. It also presents the service dependency graph model and assignment of cloud data containers to the services of a composite service in the cloud. Section 5 illustrates the experiments and provides an analysis of the experimental results, while Section 6 concludes the paper.

2. RELATED WORK

Service-oriented computing (SOC) has taken hold in cross-enterprise business settings, such as the use of FedEx and UPS shipping services in e-commerce transactions; the aggregation of hotel, car rental, and airline services by Expedia and Orbitz; or book-rating services for libraries, consumers, and bookstores. Given the widespread interest in and deployment of Web services and service-oriented architectures that are occurring in industry, the scope of SOC in business settings is predicted to expand substantially. However, there has not yet been sufficient consideration given to how services within a cloud computing environment are selected and how the selected services need to be deployed (or configured) with regard to the processing, storage, and bandwidth resources of the cloud to provide higher-level functionality and improved efficiency.

Cloud computing has recently opened another door to sharing resources in enterprise and scientific computing. It greatly reduces the start-up costs of configuring operating systems, infrastructure, and platforms and providing hardware for deploying the functions of a software system in the form of services. It follows a charge-by-usage scheme as in utility computing. Meanwhile, it relies on specially designed distributed processing capabilities to provide linear and even exponential scalability. Multiple users with very different requirements for their operating systems and platforms are provided individual and unaffected usage of resources by virtualization. Physically, one machine or a set of machines may be shared by multiple clients. Multiple physical machines or resources may be employed collectively to serve the same customer or redundantly to provide reliability and disaster recovery capacity.

Amazon's Elastic Compute Cloud (EC2) is an example of a cloud computing environment. EC2 offers flexible and on-the-fly computing capacity via a Web service. It has provided other tools for making Web-scale computing easier for developers, including a cloud-watch tool for monitoring and reporting the utilization of the resources on the cloud, scaling tools for automatically allocating EC2 instances to an application, and a load balancing service to distribute incoming requests across multiple EC2 instances [5].

Amazon Simple Storage Service (S3) is an example of a storage cloud. It provides cheap and reliable storage over the Internet. Simple Web service interfaces are provided to support basic features of data management, including write, read, and delete of objects. An object can be retrieved via a unique and developer assigned key. Developers can construct data-centric applications using either SOAP-based or REST-based interfaces. Usage on S3 is charged per storage used. For example, $0.15 per GB is charged for one month for the first 50 TB. No limitation on the number of objects is set [4].

IBM introduced ready-to-use cloud computing, named Blue Cloud, in November 2007. More than 200 researchers around the world are supporting the development of this technology. Blue Cloud includes a series of offerings that manage operations on corporate data centers, such as operating the Internet via a distrusted, network accessible fabric of resources. Blue Cloud is supposed to replace the traditional data access concepts based on local machines or remote server farms. Blue Cloud is based on IBM's Almaden Research Center cloud infrastructure. It relies on virtualized operating systems like Xen and uses the parallel workload scheduling technique of Hadoop [10] to achieve scalability. Blue Cloud uses IBM's Tivoli software to check the performance of the provisioned servers and ensure they meet service-level agreements [6].

3. MOTIVATION

Let us assume we have a simple application, called ExternalApplication1, consisting of sequential invocations of three services deployed in the cloud, as shown in Figure 1. Figure 1 additionally shows the message exchanges between the application, which is a composite service, and the three services, service 1, service 2, and service 3. We can use either of the two major protocols for compositions, i.e., Web Service Choreography Description Language (WS-CDL) and Business Process Execution Language for Web services (WSBPEL) to describe the message exchanges. We arbitrarily choose WSBPEL here for illustrative purposes. The description of the control and data links is shown in Listing 1. WS-CDL can be mapped one-to-one to WSBPEL constructs, and therefore our method is also applicable to WS-CDL.

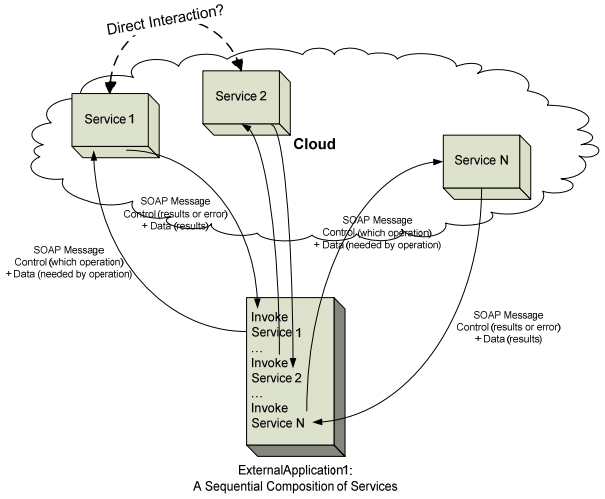

Figure 1. An example of a simple application external to a cloud and consisting of a sequential service composition

Here, ExternalApplication1 sends the first SOAP message upon seeing the first invoke construct in BPEL description. The SOAP message includes (1) control information, such as the

access point of the operation and the name of the operation, and (2) data, such as the input parameters needed by the operation. Service 1 processes the request with the given input data and returns to ExternalApplication1 (1) control information such as an indication of results or error, and (2) data such as the results. ExternalApplication1 will do the same when seeing the second and third of the "invoke" commands. In this case, we can observe that there are many messages transmitted to and from the cloud. This means that applications with a heavy data flow might be inefficient, which will impair the scalability benefit created by the cloud. Therefore, a more desirable way for such communications is via messages between the cloud-deployed services and ExternalApplication1 that uses the services to establish direct message transmissions between the services within the cloud. This way, the scalability issue can be solved by the elastic infrastructure offered by the cloud, instead of consuming communication resources at the edge of the cloud. Of course, if one service can always pass the result to the other, as in this simple example of ExternalApplication1, things will be much easier. However, this is not the case for most applications.

For better understanding, the example of ExternalApplication2 in Figure 2. It is a composite service composed of again three component services: Service 1, Service 2, and Service 3. But this time, Service 3 cannot run until it gets the inputs it needs from Service 1 and Service 2, but it expects these to come from one SOAP message, not two. So, there needs to be a container for intermediate results. Also, Services 1 and 2 must somehow know that their outputs go to Service 3. Both can be used to manage the message exchanges between component services. Meanwhile, both require something in the cloud that can store intermediate results. Also, if there is an error (e.g., Service 3 fails), then the error message must somehow get back to the original caller of Services 1 and 2.

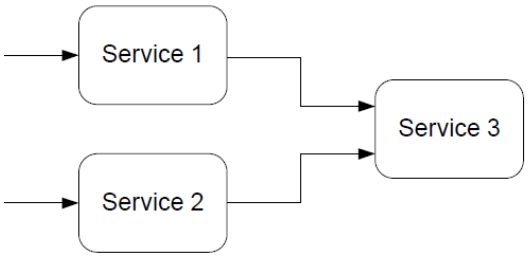

Figure 2. ExternalApplication2 composed of a parallel composite service

Listing 1.

```
<flow>...
<sequence>
    ...
    <invoke name="invocation1"
partner="service_1"

portType="asns:Serivce_1PTA"
                operation="PTAOpA"

inputContainer="containerIn"
                outputContainer="service_1-
to-process">
      <target linkName="s0-to-s1"/>
      <source linkName="s1-to-s2"/>
    </invoke>
    <invoke name="invocation2" partner="
service_2"
                portType="asns:
Serivce_2PTB"
                operation="PTBOpB"

inputContainer="containerIn"
                outputContainer="
service_2-to-process ">
      <target linkName="s1-to-s2"/>
      <source linkName="s1-to-s3 "/>
    </invoke>
    …
</sequence></flow>
```

4. CLOUD DATA CONTAINERS AND WEIGHTED SERVICE DEPENDENCY GRAPHS

In this section, we introduce the concept of cloud data containers and weighted service dependency graphs. We illustrate the container assignment scheme using these graphs.

4.1 Cloud Data Containers

Cloud data containers are external data storage systems used by one or more services deployed in a cloud. Cloud data containers hold input, output, and fault messages temporarily, as well as any related data associated with the execution of a composite service or application on multiple services in the cloud. Data containers can be located at different nodes within a cloud. An illustration of what a cloud with data containers may look like is shown in Figure 3.

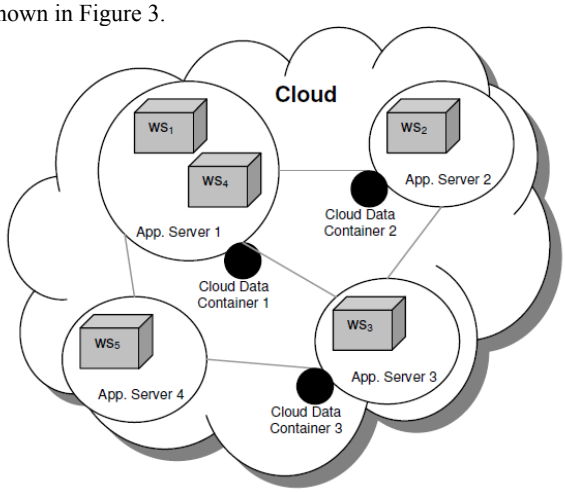

Figure 3. Cloud Data Containers

A cloud may consist of many servers, including application servers, which are shown as white circles in the figure. Each application server can host a number of services. For example, application server 1 hosts the two services WS_1 and WS_4, while application server 2 hosts one service, WS_2. Some application servers may be equipped with cloud data containers, shown as small dark circles attached to the edge of the application servers. Application servers work with the attached cloud data containers to prepare messages to be transmitted to other correlated services used by the external application.

The application servers issue instructions to write to and read from the cloud data container. The cloud data container includes the following information for each possible data exchange for an external application that involves a service hosted at the corresponding server:

1. BusinessProcessID: unique identifier for the application involving a data exchange

2. ProcessInstanceID: unique identifier for the application instance involving a data exchange
3. MessageID: unique identifier for the message exchange within a process instance
4. LocalService: unique identifier for the service that writes the content of the message to the container
5. RemoteService: unique identifier for the service that reads the content of the message from the container
6. Time-to-live: time stamp indicating the end of the message's validity period
7. ExchangeData: objects of the exchanged data
8. Size: number of bytes of the data
9. Message: content of the SOAP message

Meanwhile, the application server that hosts the last service(s) of an application will be selected as the *data exchange hub* (DEH) of the application. A DEH is an application server whose associated cloud data container maintains the process table for an external application. The process table is called the *container data directory structure* (CDDS), with each data entry recording the following information of a data flow for the external application:

1. BusinessProcessID: unique identifier for the external application involving a data exchange
2. ProcessInstanceID: unique identifier for the external application instance involving a data exchange
3. MessageID: unique identifier for the message exchange within a process instance
4. MessageTemplate: xml SOAP message with place holders for objects
5. NodeList: which nodes in the cloud are involved in the data exchange of this message, including servers of the source and remote services.
6. Time-to-live: timestamp indicating the end of the message's validity period

The DEH of the application is capable of generating SOAP message templates with place holders for actual objects. After being provided with the objects, the DEH will insert the objects into the message template to form a SOAP message. An example SOAP response message and its template for Purchase_Order are given in Figure 5. In addition, each application server is configured with a plug-in library accessible to the services, which allows the services to call and form the XML messages using the data objects in the cloud data container.

Each service has access to its own application server's data container and has knowledge of where its data objects are stored, but no knowledge of where other providers' objects may be stored. Very likely, each application server knows which objects it stores for each service deployed with itself, but not which objects other application servers may have in their data containers. Among all the data containers used by an external application, only one data container (i.e., the DEH) has knowledge of where data objects for each message of the external application are stored in the cloud.

Figure 4 demonstrates the end-to-end process of the data exchange using the cloud data containers. In order to facilitate data container access, additional APIs are provided to the services. The most important ones are "write" and "read," which allow the services to write and read the cloud data containers.

As shown in the figure, upon receiving a request from an external application, initialization will be done in the cloud, including initialization of the container data directory structure attached with the last service at the DEH, in this case, application server 3. The initialization of the CDDS includes assigning the physical cloud data containers to the required services. This is done by querying each application server for the services it is hosting. Once this information is available, the data container assignment can be decided. One straightforward way to pick a container is to always assign a container attached directly to the application server to the services. If there is more than one such data container, we can perform a load-balancing operation to select the one with the expected lowest load. However, such a container might not always be available for various reasons. For example, there might not be a container attached to this server or the container is full and cannot store more data. A more carefully planned scheme would be to select the container that is with the first server on the best path to the destination service, if there is no container at the hosting server.

This simple scheme cannot guarantee scalability in terms of the overall resource consumptions in the cloud in serving the external applications. For example, the number of containers per application may be too many and proportional to the number of services. The containers might be unused most of the time. More importantly, the bandwidth consumption might be high if a random container is selected for the services of an application server. Therefore, we have designed a more efficient container assignment scheme based on a weighted service dependency graph, which is discussed in sections 4.2 and 4.3. We take into consideration the above issues in designing the assignment scheme.

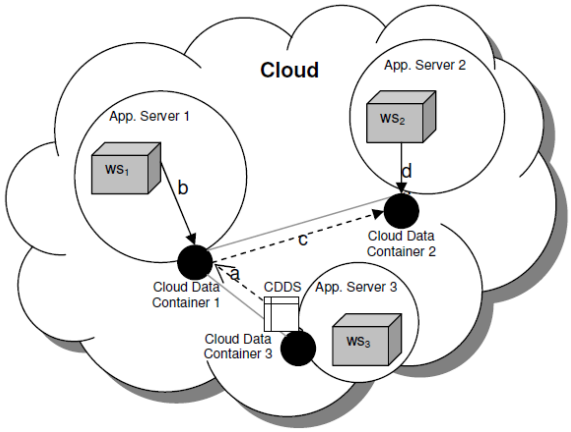

Figure 4. Process of data exchange with cloud data containers

Once the CDDS has been initialized, a multicast will be triggered to update each individual application server with the address of the data exchange involved by their hosting services. This is shown as arrow (a) in Figure 4. With this, the application server will be able to process the "write" and "read" requests from the hosted services. Arrow (a) also represents transmitting a message template with place holders to the relevant application server. After initialization, WS_1will start. Upon its completion, it will call the "write" API provided by the application server to write the objects in the output message into container 1, shown as arrow (b) in Figure 4. "Write" is asynchronous, meaning once the objects are written to container 1, WS_1 returns and does not wait until a success acknowledgement from container 2, to which the destination service is assigned. Transmitting the message to container 2 and its success are guaranteed by DEH, because it instructs and oversees packaging objects into the message template and (possibly remotely) writes to the corresponding cloud data container, in this case, container 2. This is shown as arrow (c) in Figure 4. This message will be read by WS_2 via application server 2 at an appropriate time, as shown by arrow (d).

```
<?xml version="1.0" ?>
- <soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-
    envelope"
    soap:encodingStype="http://www.w3.org/2001/12/soap-
    encoding">
  - <soap:Body xmlns:p="purchase_order">
    - <p:PayByCreditCard>
        <p:OrderNumber>dept1staff10c23</p:OrderNumber>
        <p:CreditCardNumber>1233-1265-1298-
          1253</p:CreditCardNumber>
        <p:Amount>US$150</p:Amount>
      </p:PayByCreditCard>
    </soap:Body>
  </soap:Envelope>
```

```
<?xml version="1.0" ?>
- <soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-
    envelope"
    soap:encodingStype="http://www.w3.org/2001/12/soap
    -encoding">
  - <soap:Body xmlns:p="purchase_order">
    - <p:PayByCreditCard>
        <p:OrderNumber>null</p:OrderNumber>
        <p:CreditCardNumber>null</p:CreditCardNumber>
        <p:Amount>null</p:Amount>
      </p:PayByCreditCard>
    </soap:Body>
  </soap:Envelope>
```

Figure 5. An example data exchange message and its template in SOAP

4.2 Weighted Service Dependency Graph

A service dependency graph (SDG) is a directed graph that is constructed dynamically to show all possible input-output dependencies among service operations [3][11]. There are two types of nodes in an SDG: *service operation nodes* and *data entity nodes*. Service operation nodes model operations of services and data entity nodes model their input and output attributes. Edges in an SDG link one node to another. Some edges link the nodes of the input attributes to the nodes of the corresponding service operation. For such data entity nodes, the operation nodes are referred to as consumer operation nodes. Some other edges link a node of a service operation to the corresponding nodes of its output attributes. Similarly, for such data entity nodes, the operation node is referred to as a producer operation node. *Data dependencies* in a service dependency graph are defined as the dependency relationships between the consumer operation nodes and the producer operation nodes of the same data entity nodes.

For this research, we enhance a service dependency graph by annotating the edges that connect service operation nodes to data entity nodes with weights. The weights correspond to the size of the data entity pertinent to the data exchange represented by that edge. In other words, an edge is weighted more if the data entity transmitted between the two services connected by the edge is large. Such service dependency graphs are called weighted service dependency graphs (WSDG).

A segment of a WSDG shown in Figure 6 depicts the pattern of dependency of services that process purchase orders. It encompasses functions for requesting purchase orders as well as making payments and optionally giving rebates to customers who meet certain criteria. It also accounts for delivery of the purchased items. Payment is allowed by credit card and personal checks. Both payments and rebates are referred to via an order number, which is created when the purchase order is submitted. The figure represents the pattern of input and output dependency relationships of services. For example, for "Order Number", "Request Purchase Order" is the producer operation, and "Pay By Credit Card", "Pay By Check" and "Rebate Credit Card" are the consumer operations. In other words, "Pay By Credit Card", "Pay By Check" and "Rebate Credit Card" depend on "Request Purchase Order" via "Order Number". The edge connecting "Request Purchase Order" to "Order Number" has a weight of 9, and the one connecting it to "Item Names" has a much larger weight of 50, as shown in the figure.

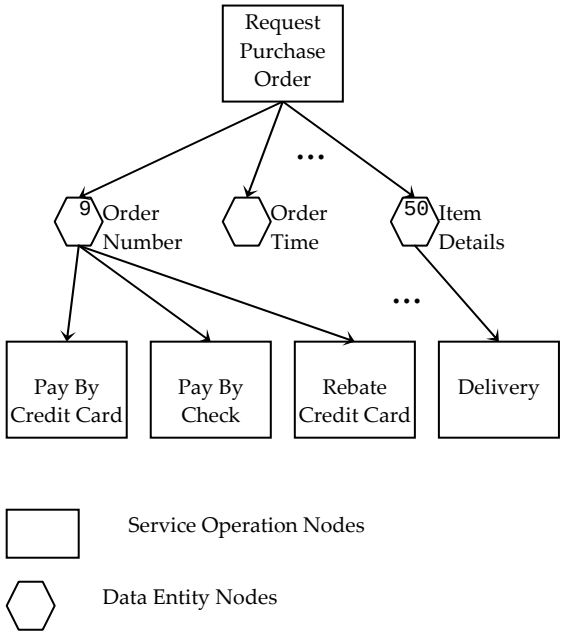

Figure 6. An example weighted service dependency graph

A WSDG can be constructed by analyzing the <partnerlinks> in the BPEL document of the external application and the WSDL documents of two Web services that are sequential, parallel, or related by other control constructs. The pseudo code in Listing 2 can be used to generate service dependency graphs from sequential structures.

Listing 2:

```
1.  for each <sequence> element in BPEL {
2.      Get next <invoke>, <receive> and <reply> element;
3.      Assign this element to irr;
4.      L1:
5.      Extract the value of <operation> of irr and assign it
        to op1;
6.      Extract the value of <portType> of irr and assign it
        to pt1;
7.      Get the next element that immediately follows irr in
```

<sequence>
8. Assign this element to *irr_next*;
9. Extract the value of <operation> of *irr_next* and assign it to *op2*;
10. Extract the value of <portType> of *irr_next* and assign it to *pt2*;
11. *if* output(*op1*) overlaps input(*op2*) {
12. Record the size of the overlapping objects to calculate the weights {*w_i*} ;
13. Construct two nodes for op1 and op2 as operation nodes and add them to the WSDG, if they are not already there;
14. Construct the corresponding data nodes for intput(p1), output(p1), input(p2) and output(p2), and add them to WSDG if they are not already there;
15. Connect the operation node(s) to the corresponding input and output data nodes by edges;
16. Denote each edge with the corresponding weight *w_i*;
17. *irr = irr_Next;*
18. Go to *L1*;
19. }
20. }

In the above pseudo code, we process the elements within the <process> construct. First, we begin with a <sequence> element and track down any <invoke>, <receive>, and <reply> elements within the <sequence> element in line 2. A sub-element of portType and operation contains the corresponding operation that it invokes on other services or is invoked by other services. So we extract both sub-elements as in line 5-6. Next, we look at the next <invoke>, <receive>, and <reply> element and we do exactly the same to extract its own <operation> and <portType> elements as in lines 7-10.

After these are done, we are able to check if any variables used in the input and output of these two consecutive Web services are the same, or if there are overlaps on some individual elements of the input and output as in line 11. If so, we record the size of the overlapped elements and calculate the weight of the data overlap given the size of the overlapping element. The weight will later be used to mark the edge connecting the data entity node and the operation node as in line 12. Then we construct nodes for the operations and input and output data entities and add them to the WSDG, if they are not already there. Edges that connect operation nodes and the corresponding data entity nodes are also constructed and also denoted with the calculated weight. Lines 13-16 in the pseudo code correspond to such activities.

This process is repeated until all the <invoke>, <receive>, and <reply> elements within the <sequence> element are traversed and all <sequence> elements are traversed as in lines 17-18.

4.3 Data Container Assignment by WSDG

In this research, we need to solve the problem of assigning data containers to services for efficient data exchanges of cloud applications. Similar problems, such as which is the most efficient route for data to travel from one place to another, have been studied in the domain of network metrics (e.g., QoS-based routing). Solutions such as Dijkstra's algorithm for single source shortest path are often used in network routing. Assigning data containers to services is a more complex problem. Here, multiple services are possibly located at different servers and they need to choose their data containers. Such a data container assigned to the service may or may not be co-located with the service itself, depending on various constraints including load balancing. Data transferred are defined by two interacting services for the purpose of completing an application. Once data are in the containers, data transfer occurs between two data containers not two services. This problem cannot be simply solved by directly applying existing solutions such as Dijkstra's algorithm.

Our purpose is simple: to define an assignment scheme for data containers while minimizing the traffic created in the cloud due to the data exchange for a particular external application using cloud data containers. The volume of traffic in our problem is determined by two factors: (1) the amount of data that is to be transmitted and (2) the length of the route during the transmission. We will explain these two factors in detail.

We use an example to illustrate the assignment scheme. In terms of the first factor of the volume of the data, we model it as the weight w_i of an edge that connects one service to the other in a WSDG. An example WSDG and its conversion are shown on the left hand side and right hand side of Figure 7. (For illustration purposes, we have removed the data entity nodes in the weighted SDG, which results in what we call a converted WSDG.) In Figure 7, three service operations, i.e., operations a, b, and c are highlighted. The weight from a to b is 9. Two weights from a to c are 6 and 5, respectively. We assume that all the objects transferred between a and c travel via the same route. In this case, the weight from a to c can be seen as an average of two weights, or 5.5.

Equation (1) shows the calculation of the weight between two service operations a and b i.e., $w(a,b)$, where $\{d\}$ represents the set of overlapping data between a and b, and n represents the total number of the overlapping data items. Equation (2) shows the calculation of the total weights between two application server nodes N_1 and N_2 , i.e., $w(N_1,N_2)$, where m denotes the number of pairs of interacting service operations in two application server nodes.

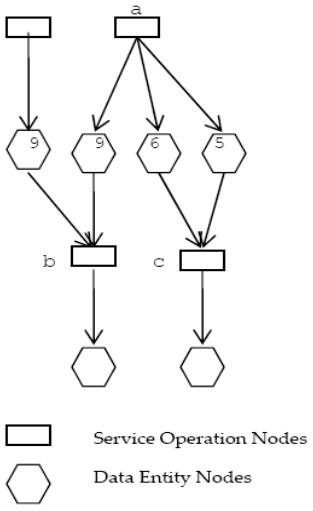

Figure 7(a). Weighted service dependency graph

$$w(a,b) = \sum_{a,b\in\{d\},i=1..n} w_i \tag{1}$$

$$w(N_1,N_2) = \frac{\sum_{a\in N_1, b\in N_2} w(a,b)}{m} \tag{2}$$

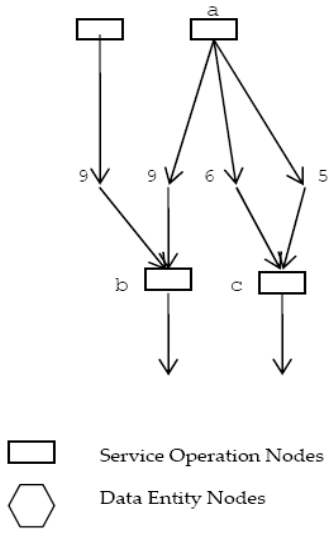

Figure 7(b). Converted weighted service dependency graph from 7(a).

In terms of the second factor of the length of the transmission, we assume that the network distance of any two application server nodes in a cloud are known in advance. The lengths between any two application servers can be represented as an m by m matrix L, where m again is the total number of application server nodes in the cloud. Equation (3) can be used to calculate the cost of using certain data containers for the application, i.e., $c_j(N_1, N_2)$. $l_j(N, N')$ returns the length of the jth path between nodes N and N', which is the element at the N th row and N' th column of L or the sum of $l(N, N'')$ and $l(N'', N')$, where N'' is an intermediate node between N and N'. $bd_j(N, N')$ is the average bandwidth of the jth route between nodes N and N'. In (3), nodes N_a and N_b correspond to where the containers for the services in N_1 and N_2 are located respectively. Equation (4) shows the calculation of the total cost contributed by all the data communications of external application *p*. Here *P* corresponds to the set of server pairs that communicate due to information exchanges required by *p*.

$$\begin{aligned} & c_j(N_1, N_2) = w(N_1, N_2) \\ & * (l_j(N_1, N_a / bd_j(N_1, N_a) \\ & + l_j(N_a, N_b) / bd_j(N_a, N_b) \\ & + l_j(N_b, N_2) / bd_j(N_b, N_2)) \end{aligned} \tag{3}$$

$$c_j(p) = \sum_{(N_i, N_k) \in P} c_j(N_i, N_k) \tag{4}$$

The following algorithm can be used to calculate the best assignment scheme of containers to the services for external applications using services in a cloud environment.

Listing 3:

---

1. **for** each pair of service operations *(a,b)* in the WSDG that exchange data for application *p*{
2. Use (1) to calculate *w(a,b);*
3. record it in {*w(a,b)*};
4. }
5. **for** each pair of application server nodes (N_i, N_k) involved in application *p* {
6. Use (2) to calculate $w(N_i, N_k)$ and record them;
7. }
8. **for** all possible assignments of containers to services in application *p*, i.e. {a} {
9. **for** all possible paths *j* between any two communicating server nodes{
10. use (3) to calculate $c_{a,j}(N_i, N_k)$ of the *jth* path for assignment a and record them;
11. }
12. use (4) to calculate the total cost $c_{a,j}(p)$ and record them;
13. }
14. *a**=min_arg($c_{a,j}(p)$);
15. return *a**;

In Listing 3, we first calculate the weights between each pair of service operations in the application that exchanges data, as in lines 2-5. Using such weights, we can calculate the weights between two services that their operations exchange data as in line 2. Services of the same application that exchange data may be deployed with different application servers. One server might have multiple pairs of services whose operations exchange data. So, one extra step is to calculate the weights between any pair of servers that exchange data and then the cost of such exchange considering different paths of transmitting data as in lines 6-9. Once all paths have been investigated, the path that has taken the minimum cost will be returned.

Here we further illustrate this assuming an example length matrix for the application server nodes in the cloud as in (5). Therefore, $l(N_a, N_b)$ =3, $l(N_a, N_c)$ = $l(N_a, N_d)$ =1, $l(N_b, N_c)$ =2, and (N_d, N_b) =1. We use the same weights shown in Figure 7. We assume that each service operation belongs to three different services, i.e., *sa, sb,* and *sc*, respectively, and each service is with a different node, respectively. There is only one communication of this application, which is between service *a* and service *b*. According to (3), we can calculate $\min c(N_a, N_b)$ = 9*1+6*2 = 21, when taking the path of a->d>b. Therefore, the best container nodes for services *sa* and *sb* will be N_a and N_b, respectively.

$$L(N_a, N_b, N_c, N_d) = \begin{matrix} 0 & 3 & 1 & 1 \\ & 0 & 2 & 1 \\ & & 0 & 3 \\ & & & 0 \end{matrix} \tag{5}$$

$$NC(s, N_1) = \forall N \begin{pmatrix} N, M \in j, \min\limits_{j} \arg c_j(N_1, N_2), \\ l(N_1, N) \le l(N_1, M) \end{pmatrix} \quad (6)$$

5. EXPERIMENTS

We simulated our container assignment scheme to analyze its effectiveness. We compare the cost of an application external to a cloud using our scheme of assigning containers, as defined in section 4.3, with the baseline scheme. The baseline scheme is defined as follows: for any Web service, select the container that is encountered first on the best path (which has the minimum cost) of any communication from the service to its corresponding destination. Formally, it can be written as in (6), where $NC(s, N_1)$ denotes the container node of service *s* on server node N_1, N_2 is the server node with which *s* communicates, and *l* is the distance between the two nodes.

In the experiments, there are two parameters, (1) *P*, the possibility that the costs of two schemes are different and (2) Δ, the amount of the cost difference, if any. The simulated cloud has 300 server nodes, each hosting a number of services. The number varies between 0 and 10 uniformly. The network connectivity of the cloud is 20%. The number of communications from one service to another and the total number of communicating services for an external application vary. We also vary the distances between two communicating services for the application. The above three facts determine, jointly, the cost of the external application.

The results of the cost savings of our scheme as the percentage of the baseline scheme is given in Figure 8. We have experimented on three sets of Δ values. The first set is for *P* = 0.2, meaning that the probability is 0.2 that there will be a cost difference. Under this circumstance, we have taken three Δ values, i.e., 10%, 20%, and 30% of the cost of the baseline scheme. This is repeated for *P* = 0.5 and *P* = 0.7. In all cases, the savings increase proportionally to the value of Δ. This means that our scheme has larger savings if the cost difference increases. Also, we see that the line corresponding to *P* = 0.7 is above the line corresponding to *P* = 0.5, which is again above the line corresponding to *P* = 0.2. This means that our scheme saves more as the probability of a cost difference increases.

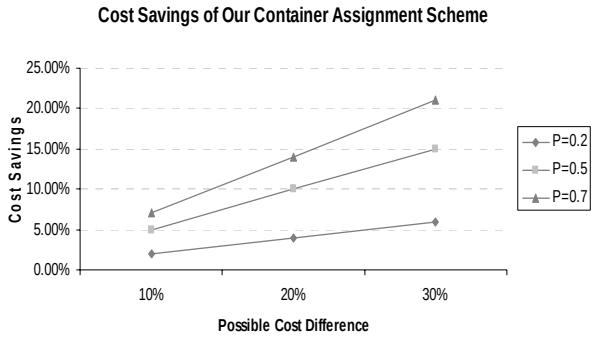

Figure 8. Cost savings using our container assignment scheme

6. CONCLUSIONS

A cloud computing environment can provide beneficial infrastructure scalability in processing, storage, and bandwidth. A benefit of services is that they can be deployed independently of each other, yet can be composed to work in concert to achieve higher-level functionality. However, when the services are moved into a cloud to take advantage of the cloud's scalability, the storage and bandwidth scalability might not be fully realized unless special provisions are made. We address the provisions in this paper.

In particular, we consider the possibly large amount of data exchanged between an external application and the services in the cloud. We have presented a technique of using transitional storage we term cloud data containers to hold immediate process data to reduce the data traffic (communication bandwidth) to and from the cloud. In essence, we use the scalability of cloud storage to reduce the communication bandwidth, reduce the time needed to process an application consisting of composed cloud services, and thereby improve throughput. The validity and efficiency of our data container assignment and selection scheme are demonstrated by experiments. The contribution of the paper is to provide a solution to the scalability issue of a cloud environment where software-as-services is deployed.

We are continuing to investigate the performance and benefits of this technique when it is applied to typical types of applications with a variety of real-world characteristics. We are also interesting in customizing this technique for catering to different types of applications.

7. REFERENCES

[1]. Ryszard Kowalczyk, Michael N. Huhns, Matthias Klusch, Zakaria Maamar, Quoc Bao Vo: Service-Oriented Computing: Agents, Semantics, and Engineering, AAMAS 2008 International Workshop, SOCASE 2008, Estoril, Portugal, May 12, 2008, Proceedings Springer 2008

[2]. Michael N. Huhns, Munindar P. Singh, "Research Directions for Service-Oriented Multiagent Systems," Internet Computing, 2005.

[3]. Q. Liang, L. N. CHAKARAPANI, S. Su, R. N. CHIKKAMAGALUR, H. Lam,"A Semi-automatic Approach to Composite Web Services Discovery, Description and Invocation", 2004, Vol. 1, No. 4, International Journal of Web Services Research (IJWSR).

[4]. http://aws.amazon.com/s3/

[5]. http://aws.amazon.com/ec2/

[6]. http://www-03.ibm.com/press/us/en/pressrelease/22613.wss

[7]. http://www.w3.org/TR/soap/

[8]. http://www.w3.org/TR/wsdl

[9]. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[10]. http://hadoop.apache.org/core/

[11]. "AND/OR Graph and Search Algorithm for Discovering Composite Web Services", by Q. LIANG, S. SU, 2005, Vol. 2, No. 4, International Journal of Web Services Research (IJWSR), page 46-64