AN ALGORITHM TO SELECT THE OPTIMAL COMPOSITION OF THE SERVICES

[1]MOHAMMAD K. SEPEHRIFAR, [2]KAMRAN ZAMANIFAR, [3]MOHAMMAD B. SEPEHRIFAR

[1]M.Sc. Student, Department of Computer Science and Engineering, University of Isfahan, Iran

[2]Asstt. Prof., Department of Computer Science and Engineering, University of Isfahan, Iran

[3]Asstt. Prof., Department of Mathematics, University of Mississippi, USA

E-mail: mksepehrifar@eng.ui.ac.ir, zamanifar@eng.ui.ac.ir, moe@olemiss.edu

Abstract

Because of providing services to the users in the heterogeneous distributed environments, service oriented systems are very important. Most of the time, the individual services do not have the sufficient conditions to provide any services to the users. In order to resolve the aforementioned problem, one may compose several individual services together.

In this paper, we propose a dynamic approach to select the best composition. This composition is selected based on the quality and the compose-ability of participated services.

In the real life situation, we are facing with different servers, in which they present the same services with the different interfaces. This proposed method considers these varieties. Another advantage of the proposed approach in the paper is to recognize the feasibility of the composition process at any point of execution.

We introduce an algorithm with a better speedup and a less consumption of memory to select composition of services dynamically. Needless to say, that this algorithm considers matching of interfaces of services.

Keywords: *Dynamic Service Composition, Composed Service, Quality of Service (QoS), Optimal Selection, Interface Matching*

1   INTRODUCTION

In the recent years, the complexity of software systems has become one of the main reasons developing the distributed systems. Service-oriented systems are one of the most applicable distributed systems. These systems provide user's requirements through cooperation with a set of distributed services inside of the networks such as internet [1], [2].

One of the main problems in the field of service-oriented systems is how to compose services and how to manage this composition. For doing the composition task, there are many different procedures in the literature [3]-[6]. However, selecting the optimal composition from different compositions still is an open problem [2]. In this paper, we address a solution for this problem.

Section 2 studies the service oriented systems and its related definitions. These definitions will be used the next sections. In section 3, we introduce a method to select the optimal composition of services, dynamically. Section 4 shows the results derived by a simulation study.

2   COMPOSING OF SERVICES

Service oriented systems are functioning based on services. Composing of the services can make a value added service.

These services provide service to the clients (A person or a software module that gets benefits from services of servers). A client may find its appropriate services from the metadata information of the services.

Of course, by increasing the number of services and by expanding the space of system environment, it is better to register the services through a service

broker. Moreover, having a more speed and a better property of services are the reasons that a client may use a service broker.

Each service has three properties as follows: the quality, the interface and the functionality.

The quality of each service depends on functional and non-functional requirements. In a service, finding a proper output is called the functional requirements. While the speed of the computation of finding this output is called a non-functional requirement. Based on some criteria, a client gives score to each satisfied requirements. As the result, this client may accept the immediate first output or wait for a longer time to find a more proper output.

In order to compare the quality of service, we need to transfer the quality of the service into the quantity values. We measure the total quality of service (QoS) by using the total points that a client assigns to each satisfied requirement. We calculate this quantity (QoS) as follows: [7]

$$QoS(s) = \sum w_i * Score_i(s)$$

In the above formula, client assigns score w_i to the ith satisfied requirement and service s satisfies this requirement with quality $Score_i$.

Interface is another property of the services. Each service has two types of interfaces: Input and output. Input interfaces indicate the input parameters and output interfaces indicate the output parameters. Two services are called match if output parameters of one of them is the same as the input parameters of the others. A community is a set of services, which are matched together.

In order to increase the usability of services most of the time we design services in such a way that the functionality property of service could solve the problems, which are simple and basic. Software developers select the proper composition of services to solve problems that are more complex. By executing each of these compositions through a specific process, they can reach to the

solution. This process is determined during execution plan associated to the problem.

2-1 Execution plan

Compound task is a set of tasks to solve a specific problem. In order to execute a compound task one has to execute a set of tasks in the specific order. In an execution plan, we show all tasks related to the compound task with their assigned position.

In the travel case study, we want to have a correct and an optimal planning. To propose a set of related selections, we need to have an organization between different schedules.

In this problem, execution plan is designed in such a way that one could travel between the two points, the staying and visiting the destination point and also be able to return. The execution plan for travel problem is called itinerary. This itinerary is designed based on the conditions of the traveler and the environment. [8], [9]

Figure 1 illustrates part of this plan. In this graph, tasks t_1, t_2, …, t_5 and the process of the execution between these tasks are shown (for example task t_5 is executed when task t_4 is accomplished). The fork streaming represents a condition choice.
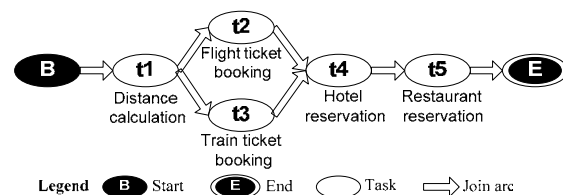

Figure 1: execution plan of travel problem

Depends on how many paths from the first task to the last one involved, there are two types of execution plans: the simple pipeline plan and the complex plan.

The simple pipeline is the simplest execution plan structured with only one path. The complex plan has several paths; each path can be considered as a simple pipeline. In Figure 1, the execution plan is a complex plan.

The service selection for the simple pipeline is easier than that of complex plan. After getting the optimal execution plans of these single pipelines, we aggregate them into an overall composition execution plan. In this paper, we specifically discuss how to get the optimum execution plan of a single pipeline path.

By using methods introduced in [10] and [11], we can divide a complex composition into several single pipelines. This makes the process of optimization much easier. Figure 2 shows the two pipeline paths from Figure 1.
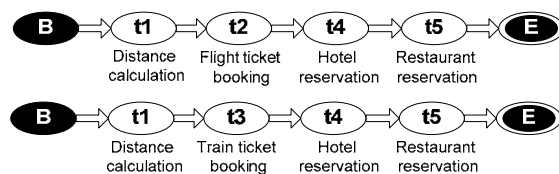

Figure 2: Complex pipeline divided into simple pipelines

After the selecting the optimal execution plan for each simple pipeline, we need to combine such plans in order to get an overall execution plan. We can use the approach mentioned in [4], [10] and [11] to combine execution plans into one overall execution plan.

We are looking for the execution of a compound task through execution of a set of services. To do so, we need to design the execution plan ([12], [13]) and then we may select the proper service for the individual tasks. In the system environment and for each task, there might be several services. We label these services as the candidate services. Thus, the execution of the candidate services associate with the plan leads into the executing of the composed service.

2-2 Composed service

In the system environment and for each task there are many different candidate services, which are provided by different providers. Selecting a proper service from these services is a difficult and sometimes impossible task. [8], [3] Automatic service composition is one solution to this problem. During the execution time, it means that one of the components of the service-oriented system does selecting and composition of services.

For each task in execution plan, we select a service in which its interface is matched with its adjacent services. Furthermore, the functionality of the service shows its ability in doing the task.

In addition to the above properties, the selected service must have the highest quality score (the highest QoS) among the set of candidate services. Finally, given these criteria, one may have an optimized composed service.

We can consider the optimization process as the local or global. In the local optimization approach, we only select valid candidate nodes in which they have the highest QoS value at each task (greedy method). Then we can get a path that represents the execution plan of services composition. However, the path created by local optimization may not be the heaviest path. That means the overall QoS value of the execution plan, which formed by services represented by nodes in the path, may not be the global optimum execution plan. We may adopt the global optimization to select the execution plan. Needless to say, that overall QoS of a composed service is calculated by sum of QoS attribute of participant services.

To select the optimal composition, One may either doing search among all services in the design time (static approach) or, as the first step, just searching for services of the current task and after finding the candidate services for this stage we go on to the next task (dynamic approach). Dynamic selections are near to real life problems, because the quality and the quantity of services are changing frequently. [14]

In the next section, we introduce a method for dynamic selection of the optimal composite service.

3 SELECTION OF OPTIMAL COMPOSITION

Based on designed execution plan, we must find a service for each task in runtime that finally have the best possible composite service.

Therefore, we introduce a method that considers each service as a graph node. With attention to order of tasks in execution plan, we

connect nodes as the edge of the graph. After we create the graph, by searching in the existed paths we can find the best path. The set of services of that path declares as the best composition. In 3-1this section, we study how to create the graph and how to do searching in the created graph.

3-1 Graph creation

All the proper services for each task in the execution plan are requested from the service broker. The service broker presents these services in the form of different communities. Figure 3 shows an instance of the execution plan. Tasks are presented as ellipses on the top of this figure and the related candidate services are shown in column under each task. Services in a community are besieged in the same rectangles.

As mentioned earlier, we create the nodes of the graph from some of the services. To connect these nodes we use the weighted directed edges of the graph. In Figure 3, the label near each edge is the weight of that edge respectively.
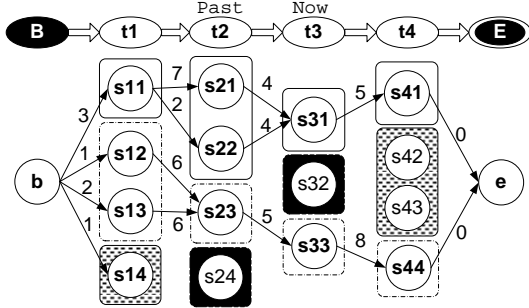

Figure 3: Execution plan and candidate services

Algorithm 1 introduces the graph creation operations completely. In this algorithm and for each task, first we receive all candidate services. Then we insert these services into a list in form of the sets of the same community services. We name this list as **Now** list (For example, the broker introduces services s31, s32, s33 for task t3 in Figure 3).

In the next step, we study each community in the created list to figure out whether it has any matched services in the list of previous task (Which is called **Past**) or not. If such any services exist (e.g. community of s31), we connect all services in

the two communities together two-by-two (e.g. edges (s21, s31) and (s22, s31)). Then we assign the quality of destination service (that is in the **Now** list) to these new edges. In order to avoid the superfluous searches, we delete that community from the **Past** list.

If we cannot find any proper community for a service in the list of previous task (e.g. the community that has services s42 and s43 in Figure 3), we delete that community from the list of studying task. The reason for this deletion is that we do not need to search this community again, when we start studying the next task. These deletions lead decreasing of additional searches so that we can create and search graph with a better speed.

```
Past=NULL; Now=NULL;
create_node(nb,B); create_node(nf,E);
foreach n in Tasks do{
   Past=Now; Now= get_cm(TRn);
   foreach CM in Now do{
      if n=1 then CMP= {B}
      else CMP= match(CM,Past);
      if CMP is not NULL then{
         foreach s in CM do{
            create_node(ns,s);
            foreach sp in CMP do
               add_edge(nsp,ns,Q(s));
            if n=k-1 then
               add_edge(ns,nf,0);
         }
         remove(CMP,Past);
      } else
         remove(CM,Now);
   }
   if Now is NULL then
   throw("It is not feasible!");
}
```

Algorithm 1: Graph creation

In summary, at the beginning of the execution for task n, the Now list has all candidate services related to this list. Moreover, the Past list has services from task n-1 that has at least one matched

service in the services of task n-2.1 For example, if we are at the beginning of the execution for task t3, the Now list has services s31, s32 and s33 and the Past list has services s21, s22 and s23. (at this point of execution, we do not know any services related to the next tasks).

Hence, we do not create any graph node from services that have not any matched services in the previous task. This operation leads to the less memory consumption as well. After executing the algorithm, the final graph has all drawn edges and services in Figure 3 (as nodes) except services s24, s32, s42 and s43.

In some situation, however, we may not be able to create any composed service from the services in the system environment. In this case, the **Now** list would be empty at the end of the execution for that task. Therefore, we may inform the user that the composition is not feasible and then we terminate the execution.

In the Algorithm 1, each function is described with these operations:

- **create_node(nodeName,service):** Creates a graph node with assigned **service** and called it as **nodeName**.
- **get_cm(TR$_n$):** It requests all services from service broker that can do the task *n*. Service broker places all that services as a set of communities of services and deliver to the requester.
- **match(CM,Past):** It searches in the **Past** list and return the community that is matched with **CM**. if there are no such community, returns **NULL** as the output.
- **add_edge(node1,node2,Q):** Connects a directed edge from **node1** to **node2** and determines its weight as **Q**.
- **remove(community,list):** Removes the **community** from the **list**.

---

[1] In the Past list, we have services of task *n-1*, in which they can be accessed via first node of the graph (with at least one path).

- **throw(message):** Terminates execution of the algorithm with showing proper **message** to the user.

After executing the Algorithm 1, a graph with multiple paths is created. Only some of these are a path from the service of the first task to the service of the last task. We name these paths the execution path, which connect the first node of the graph to the last one (e.g. nodes b and e in Figure 3).

We also label each set of services on the execution paths of the graph as the desired composed service. In order to reach to the best composition, we should select the best execution path. This can be done by doing a search in the graph to find the best path (the heaviest path).

3-2 Search in the graph

Overall quality of composed service is equal to sum of quality of its participant services. In searching a graph, one may select the composition of several services that its sum is greater than the others are. The paths with edges that have greater weight will have services with a greater quality. In order to find the best composite service, we select heaviest path from the execution paths by executing a search algorithm.

Exhaustive search algorithm is a straightforward algoritm which computs the QoS value for each possible execution plan. This algorithm selects the best one from them. Surely, it always produces the optimal execution plan. The disadvantage of the search algorithm is the time and memory consuming.

Several approaches have been presented to solve the problem of selecting the heaviest path of graph. Yan Gao *et. al* [15] proposes a six-step approach with proper complexity. That approach uses dynamic programming to solve the problem.

After accomplishing those steps, we get the heaviest path of a graph. Services that are represented by the nodes in the heaviest path will construct a global optimum execution plan of a simple pipeline services composition. In our example, after searching the graph in Figure 3, the

path {*b, s13, s23, s33, s44, e*} is introduced as the selected execution plan.

We may combine such execution plans of each simple pipeline to get an overall execution plan of the composition.

4   SIMULATION AND IMPLEMENTATION

In this section, we offer simulation study on the implementation of the Algorithm 1 with a real life example. Consider a system with Intel Pentium[©] IV processor[2] and 256 MB RAM. Also, consider the C# programming language for this implementation. We implement the Algorithm 1 on the above system and then execute it on Microsoft XP[©] operating system. We consider services with random properties as input.

In addition to the all improvements that we mentioned in the earlier sections, the proposed approach does huge improvements on the time and the memory consumption compare to the other methods in the literature (e.g. [15] and [16]). As an experimental evaluation, we implement both the proposed approach in this paper and the latest work in this area ([16]). To do this comparison, for both of approaches, we calculate the time and the memory consumption with the same input at the execution time.

In order to compare time and memory consumption of these approaches, we may consider the varieties in the number of tasks, the number of services and the number of types of service interfaces during the time. The time of algorithm execution shows the time consuming and the number of nodes in the graph which determines the memory consumption. We calculate these two values for different executions with considering the variation of one of those varieties.

 These results are illustrated in Figure 5 to Figure 7. Chart "a" in each figure shows the execution times of two algorithms. Chart "b" shows the required memory for their execution respectively; it means that in each chart, the top

curve diagram is related to the latest method in the literature and the bottom curve is related to the proposed approach in this paper (Figure 4 shows the legend of these charts).
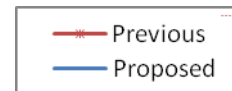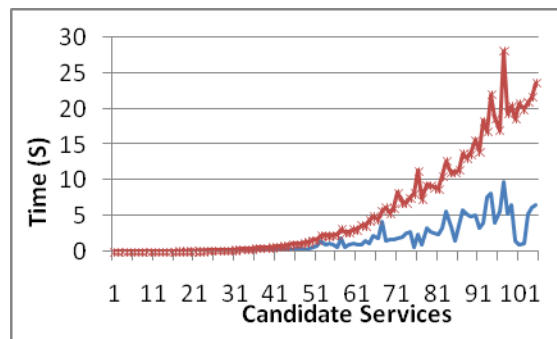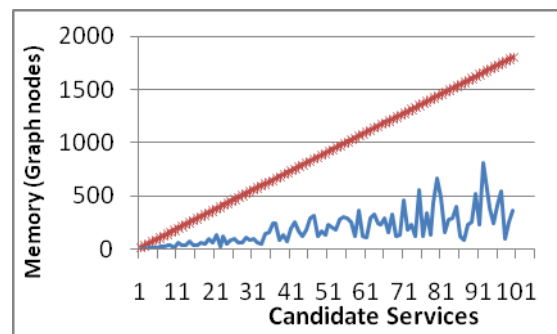

Figure 4: Legend of the charts

In Figure 5, we consider an execution plan with twenty tasks in which the number of candidate services is variable. We randomly select the type of interface for each service from ten available types.
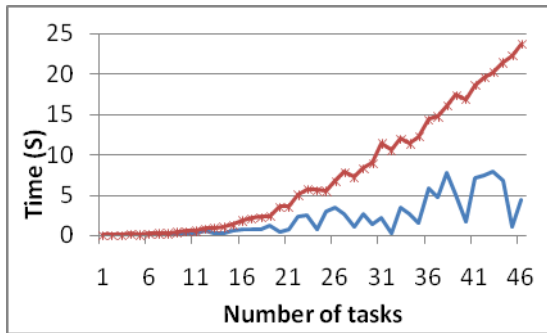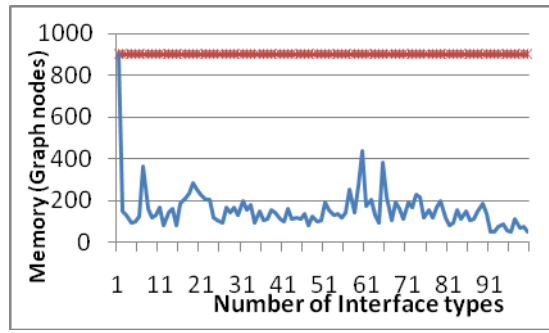

(a)

(b)

Figure 5: variety of number of candidate services

In Figure 6, the number of tasks is variable. For each task, we consider fifty candidate services. These candidates choose their interfaces from ten available types.

---

[2] Properties of the Celeron[®] D: 2.4 GH, 32 bit, 256 KB Cache memory
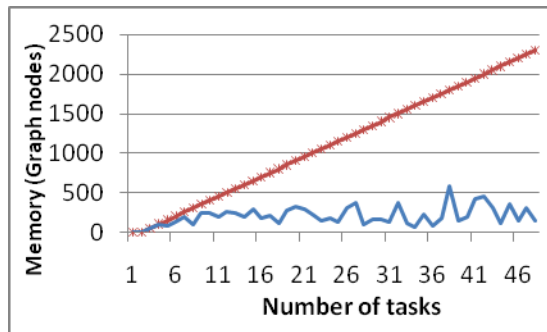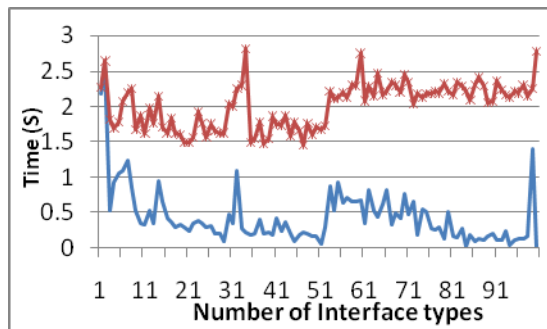
(a)

(b)

Figure 7: variety of type of interfaces

(b)

Figure 6: variety of number of tasks

In Figure 7, we consider an execution plan with twenty tasks and fifty candidate services. Interface of each service is selected from a variety number of types.

(a)

5 CONCLUSION

In this study, we proposed an algorithm for selecting an optimal composition of services dynamically. In this algorithm, we first create a graph from candidate services based on the execution plan (with attention to their interfaces). Then we label the best path in the graph as the optimum composed service.

Considering variety of service interfaces makes outputs of this algorithm more close to the real life situation. This algorithm shows the feasibility of the composition process at any point of the execution.

In the simulated study, the experimental evaluations show this algorithm has a better speed and use the less memory space compare with other similar approaches. The speedup in the creation and the searching of graph leads to the overall speedup of the execution. The aforementioned improvements decrease the waste of the resources.

6 REFERENCES

[1] Mike P., Papazoglou and Willem-Jan van den, Heuvel., "Service oriented architectures approaches, technologies and research issues." s.l. : The VLDB, Springer-Verlag, 2007, Issue 16, pp. 389-415.

[2] papazoglou, et al., "service-oriented computing: state of the art and research

challenges." s.l. : IEEE Computer society, Nov. 2007, pp. 38-45.

[3] Budak, Arpinar, et al., "Ontology-Driven Web Services Composition Platform." s.l. : IEEE International Conference on E-Commerce Technology (CEC'04), 2004.

[4] L., Zeng, et al., "Quality Driven Web Services Composition." s.l. : 12th Int'l Conf,World Wide Web(WWW), 2003.

[5] Jinghai, Rao and Xiaomeng, Su., "A Survey of Automated Web Service Composition Methods." s.l. : ICAPS, 2005.

[6] Biplav, Srivastava and Jana, Koehler., "Web Service Composition - Current Solutions and Open Problems." s.l. : ICAPS, 2004.

[7] Benatallah, B., Sheng, Q. Z. and Dumas, M., "The Self-Serv Environment for Web Services Composition." s.l. : IEEE Internet Computing, 2003, Issue 1, Vol. 7, pp. 40 - 48.

[8] Ambite, J. L., et al., "Getting from Here to There: Interactive Planning and Agent Execution for Optimizing Travel." s.l. : Proceedings of the Fourteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-2002), 2002.

[9] Knoblock, Craig A., "Building Software Agents for Planning, Monitoring, and Optimizing Travel." s.l. : ENTER, 2004.

[10] Zeng, Liangzhao, et al., "QoS-Aware Middleware for Web Services Composition." s.l. : IEEE Transactions on Software Engineering, 2004, Issue 5, Vol. 30, pp. 311-327.

[11] Yu, Tao and Lin, Kwei-Jay., "Service Selection Algorithms for Web Services with End-to-end QoS Constraints." s.l. : Journal of Information Systems and e-Business Management, 2005, Issue 2, Vol. 3, pp. 103-126.

[12] Sheshagiri, Mithun, desJardins, Marie and Finin, Timothy., "A Planner for Composing Services Described in DAML-S." s.l. : ICAPS, 2003.

[13] Sirin, Evren and Parsia, Bijan., "Planning for Semantic Web Services." s.l. : Semantic Web Conference (ISWC), 2004.

[14] Fluegge, Matthias, et al., "Challenges and Techniques on the Road to Dynamically Compose Web Services." s.l. : ICWE'06, ACM, 2007.

[15] Gao, Yan, et al., "Optimal Web Services Selection Using Dynamic Programming." s.l. : Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06) , 2006.

[16] Gao, Yan, et al., "Optimal Selection of Web Services for Composition Based on Interface-Matching and Weighted Multistage Graph." s.l. : Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'05) IEEE, 2005.