Improving Performance for Decentralized Execution of Composite Web Services

Xitong Kang, Xudong Liu, Hailong Sun, Yanjiu Huang, and Chao Zhou
School of Computer Science and Engineering
Beihang University
Beijing, China
{kangxt, liuxd, sunhl, huangyj, zhouchao}@act.buaa.edu.cn

***Abstract*—Decentralized orchestration of composite web services offers performance improvements in terms of increased throughput and lower response time. However, most relevant research literature in decentralizing service composition omit the step of selecting component services, the locations of which have major impact on the amount of dataflow messages and the volume of network traffic during decentralized execution. In order to reduce the network traffic generated by data flow messages thus to improve the performance of the decentralized orchestration, this paper presents an approach to component service selection using data dependency graphs. The performance evaluation concludes that a substantial decrease in network traffic by use of our service selection approach results in a reduction of approximately 30% in execution time on average.**

Keywords- composite web servie, decentralized orchestration, data dependency, network traffic, service selection

I. INTRODUCTION

Service composition is a widely-promoted standard to develop applications using loosely coupled and distributed web services over the Internet, and the performance of its execution is inevitably a critical issue to deal with.

The relevant research literature [1, 2, 7, 6, 5, 3, 4] confirms that enabling decentralized execution of composite service is an effective way to tackle the performance bottleneck caused by the centralized manner that is typically utilized. Most techniques dealing with decentralized orchestration is based on the approach by converting the composite service presented by a process specification into decentralized partitions, each of which associate with a component service and act as a proxy for its invocation and data transmit. By this means, component services can exchange data flow messages directly with one another in a p2p fashion, avoiding the need to pass large quantities of intermediate data back and forth through a centralized server. Experimental analysis [1, 2, 7, 3, 5, 4] suggests that decentralized orchestration brings performance benefits for reduced network traffic, increased throughput and lower response time, especially in processes where large data volumes are transported (e.j. large-scale or data-intensive workflows).

Although the relevant approaches offer such performance improvements, most of them omit the step of selecting component services, the result of which have significant impact on the network traffic volume during decentralized execution. Since the data exchanges between component services generate network traffic only when they are located at different sites, to situate component services at the same site if they have data dependence relations through the service selection may reduce the amount network traffic for decentralized execution. To get a better understanding of how service selection could affect the performance of decentralized execution, let us consider the following example.

Problem statement. Fig. 1 illustrates a composite web service of Ticket Booking. The process connects to the Enterprise service to retrieve meeting notes, next it uses an address service (Address1) to identify the formal address of the meeting place to be used by the Train service. In parallel, the application uses a Position service to identify its current position and it converts the current position to the same formal representation by using another address service (Address2). With the two addresses, the train service uses the hour of meeting notes to provide a corresponding train and the process ends with the accomplishment of payment.

A possible solution of decentralization for the composite service is presented in Fig. 2(a). Our architecture is based on a p2p service overlay where service containers are deployed at different Internet locations. Individual service providers could deploy their services at these containers. For each

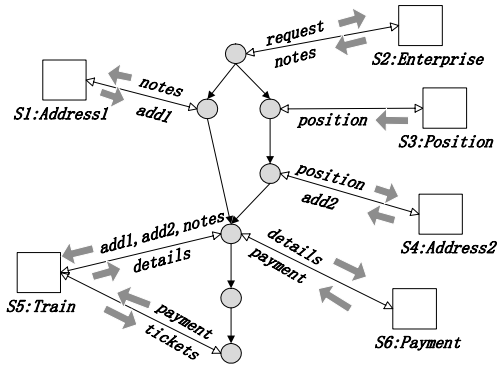
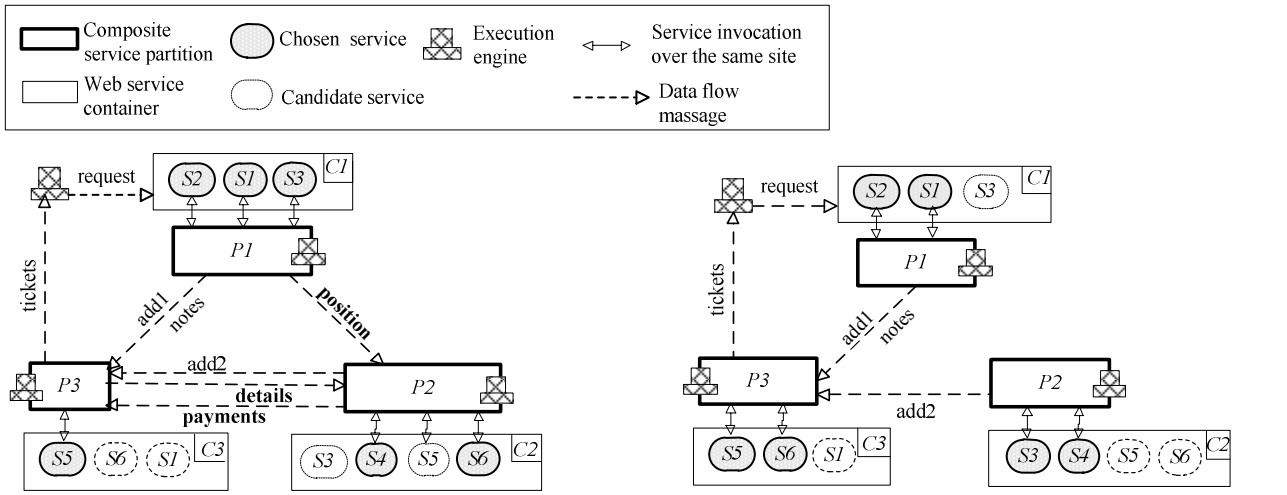


Figure 1. Ticket Booking example

978-0-7695-4129-7/10 $26.00 © 2010 IEEE
DOI 10.1109/SERVICES.2010.38

service, there are multiple instances for it that provides equivalent functionality. For example in Fig. 2(a), *S3* has two equivalent instances at *C1* and *C2* respectively, and the one at *C1* is chosen as component service. Furthermore, it requires an execution engine placed at each site. For a service composition, each sub-set of component services that are located at the same site is associated by one partition consisted of a portion of logic invoking them and executed by the engine locally. Partitions exchange data flow messages with one another in a p2p fashion. For example in Fig. 2(a), *P2* act as a proxy for the invocation of *S4* and *S6* at C2, and is executed by the engine collocated with *C2*.



(a) A Service Selection Solution for Decentralized Execution (b) Another Service Selection Solution for Decentralized Execution

Figure 2. Two possible solution for decentralized execution of the Ticket Booking example

Now consider the same example in Fig. 2(b), which give another possible solution of decentralization determined by a different service selection result. It can be seen that the amount of data flow messages represented by the dashed wire is lower than that in 2(a), i.e. the network traffic generated by the messages across sites is less in 2(b). In Fig. 2(a) for example, the **position** information produced by *S3* and needed by *S4* must be sent by *P1* to *P2* via a message across servers. However in 2(b), this data exchange do not generate network traffic, since the chosen instance of *S3* and *S4* are located at the same site (*C2*). They communicate just through partition P2 collocated with them and thus eliminate a data transmit across sites. Similarly, choosing S5 and S6 from the same site (*C3*) also eliminate a data flow message to transform **detail** and **payments** information across network. In short, the decentralization solution in Fig. 2(b) is more communication-efficient for lower amount of network traffic.

Objective. Our objective is to implement a decentralized solution for a composite service that generates relatively minimized network traffic. This is solved by a service se lection method which make the total number data exchange between services across different sites is as less as possible.

Contributions. Major contributions of this paper are as follows. We introduce a component service selection algorithm SBD, which select qualified service instances that have data dependence relations from the same site if possible, in order to reduce the data flow messages across different sites which generate network traffic. Our experiment results show significant benefits from the SBD algorithm for our example composite service and 1000 simulated service compositions. For the same hardware resource, the decentralized orchestration determined by SBD performed better than the ones determined by the random method across a range of parameters for message sizes and service instance number. The performance analysis shows that the decrease in communication overhead by use of SBD results in a reduction of approximately 30% in execution time on average.

II. COMPONENT SERVICE SELECTION FOR DECENTRALIZED EXECUTION

In this section, we present our methodology for selecting component service across network in order to reduce the network traffic of decentralized execution. Before explaining the steps to component service selection, we consider some preliminary definitions.

A composite web service is primarily presented by a process. We use a graph based formalism throughout this paper to specify the process instead of any specification languages just for clarification reason.

Definition 1(Process) *A process, P is a tuple (A, D, E, O, S) where A is a set of activities, D is the set of data shared amongst activities, O is the set of operations on data. E is a set of control flows edges with* $E \subset A \times A$ *and S is a set of web services invoked by the process activities.*

A process activity $a \in A$ could consist of a interaction with a component service via the invocation of its operations. There are also activities only consisting of some data

operations representing some business logic or complex computations, without making service invocation (e.j. conditional activity or script task in BPMN [21], etc). To present our approach easily, those activities with service invocations are designated as *fixed activities*[2]which must be run at the site of associated web services; all other activities are designated as *portable activities*[2] that can be run at any site. These portable activities should be assigned to a partition to be executed in process of component service selection, which we will detail in section 2.3.

With the definition above, we convert a process to a control flow graph *CFG(A,E)*, where each vertex represents an activity and each edge represents a control flow. Next we give an overview of the basic mechanisms of our method for choosing service instances for a composite service and decentralizing its execution.

A. Overview of Component Service Selection and Decentralized Execution

The aim of our service selection method is to determine a communication-efficient decentralization with relatively minimized network traffic in order to optimize the performance of execution. The approach consists of three steps each explained in detail in the next sections, and guided by an illustrative example (See Fig. 3). First, the data dependencies between activities are analyzed which is used as the input of component service selection. Next, component services are selected and the site where each portable activity will be executed are determined according to data dependency relationship. Furthermore, we decentralize the composite service using the criteria that each subset of component services located at the same site is associated to one partition, which is executed by the engine collocated with them.

To get a better understanding of our approach, let us consider the example depicted in Fig. 3. It gives the *CFG* of a biomolecular computing workflow which provide the function of mapping microarray data onto metabolic pathways (actually a fragment of the Taverna[13] workflow in [12]). It is implemented based on a service composition consisted of eight components from S1 to S8. We use the convention that fixed activities are represented by rectangular boxes and portable activities are represented by round boxes. The "invoke" operation in a fixed activity represent an invocation to the associated component service. The read and write operations on data items of are given in Fig. 3. We only focus on the data dependent relationship between them while the details of these services are omitted due to space constraint.

B. Building Data Dependency Graph

The first step of our approach is computing the Data Dependency Graph of a service composition presenting the direct dependence relationships between process activities which will be used in our service selection method. Since

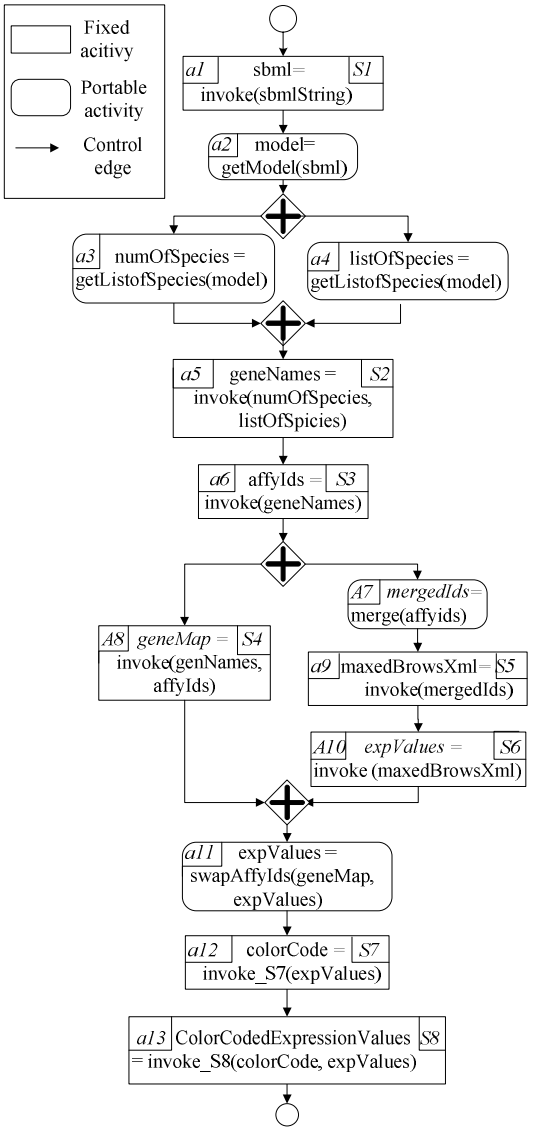


Figure 3. Illustrative example

the data dependency relationships is not defined directly in the process specification, we should analyze the usage of process data in order to extract the data flow information of process activities used for building the Data Dependency Graph.

In order to present our approach precisely, we now define the data flow relationships between activities we need. In a process specification, an activity has a *data definition* for a data item if and only if it modifies the data. For example in BPMN [11], there is a *data definition* in an activity which writes the return value of a service invocation to a variable. There is a *data use* if an activity references a data item without modifying it. For example, a conditional expression in gateway[11] activity. In order to introduce the method of computing data dependencies more easily, several sets of data flow information will be defined.

Definition 2 (reaching definition): A definition d of *activity is said to reach activity if*

1 a_j *is an successor of* a_i *in CFG(A, E)*

2 There are at least one path from a_i *to* a_j *which does not contain an activity having an redefinition of the same data item.*

For example in Fig. 3, the definition "geneMap = invoke (genNames, affyIds)" for data "geneMap" in activity *a10* reach both *a11* and *a12*. However, the operation in *a11* re-writing the data expValues kill the definition for expValues in *a6*, so the latter does not reach *a12***.** To acquire and describe the data dependency formally between activities of a process, we apply the def-use relationship into activities.

Definition 3 (def-use): *Two activities* a_i *and* a_j *satisfy the def-use relationship, if there is a definition of data item x in activity* a_i *that reach activity* a_j*, and there is at least one data use of x in* a_j

For example, there is a def-use chain about the data expValues between activity a11 and a13 in Fig. 3.With def-use relationships, we could give the definition of the data dependency graph.

Definition 4 (Data dependency Graph) *The Data Dependency Graph DDG(V,* E_d*) is a directed graph with the node set* $V \subset A$ *and the edge set* $E_d \subset V \times V$*. If* a_i *and* a_j *in V satisfy the def-use relationship, then there is an edge from* a_i *to* a_j *in* E_d.

Consequently, in order to build the DDG, we should determine the set of def-use chains. To achieve this, we should first compute the set of reaching definitions for each activity recorded in the *In set* using the data flow analysis algorithm in [8]. Then the def-use chains could be found and DDG

Algorithm1: Data Dependency Graph building
Input: *CFG(A,E),* In[a] for each activity *a*
Output: The data dependency graph *DDG*(*V*, E_d)
Begin
1 Create DDG(*V*, E_d) where *V*=∅ *and* E_d=∅
2 **for** *each activity a* ∈*A*
3 **do** *use[a]*←*statementParser(a)*//get the set of data
4 //used in activity *a*
5 **if** *usa[a]* ≠ ∅ **then**
6 *DDG. addNode(a)*
7 **for** *each data x in use[a]*
8 **do** d_x← *In[a].get(x)*//*get definition for x in a*
9 Activity *v* ← d_x.activity
10 **if** *v* ∉ *V* **then**
11 *DDG.addNode(v)*
12 **end if**
13 **if** *edge(u, v)* ∉ E_d **then**
14 *DDG.addEdge(a, v)*
15 **end if**
16 **end for**
17 **end if**
*18***end for**
end

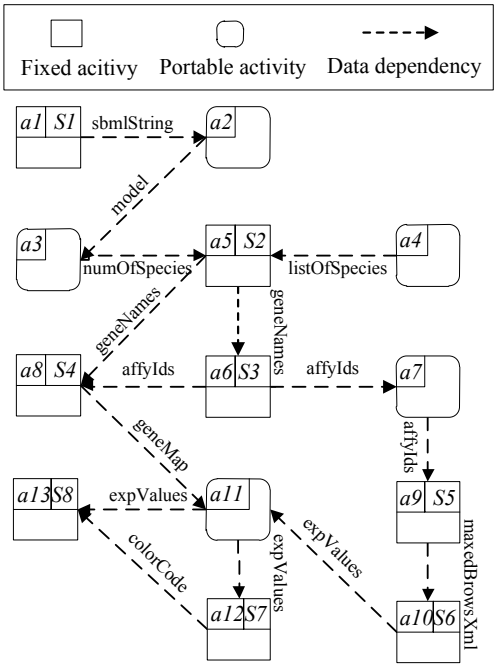


Figure 4. Data dependency graph DDG

could be built using algorithm 1, which find the def-use chains through searching the CFG. The *DDG* of a process could contain several connected sub-graphs disconnecting from each other. Fig. 4 resumes the data dependency graph between process activities introduced by Fig. 3.

C. Component Service Selection

With Data Dependency Graph of the process, we discuss further the approach of selecting component services in order to get a decentralization solution for a service composition which generates reduced network traffic. In general, the service overlay network could contain huge amounts service instances and there is exponential number of ways to choose component services. For example in Fig. 3, if each service has on average 4 equivalent instances distributed at a distinct node, there would be 4^8= 65536 possible options for service selection. Hence, an exhaustive search algorithm that tries every possible solution to yield the optimal decentralization with the least amount of network traffic is intractable, especially when the scale of composite service or the number of service instances is large.

Therefore, we give a heuristic algorithm called the SBD (Select By Dependency). The aim of the SBD algorithm is to make the total number data exchanges (data flow messages) between chosen services across different sites is as less as possible, which will yield relatively minimized network traffic during decentralized execution. The algorithm is based on the following idea: if there is a data dependence relationship between two services, then choose qualified instances for them from the same site if possible since the data exchange over the same site does not generate network

traffic. The data dependence relation between services includes direct and indirect dependency. The direct dependence relation exists between two services when the fixed activities invoking them are adjacent in the DDG. Considering two fixed activities that is not adjacent in the DDG, if there is a path (without consideration of edges' direction) between them and the activities in this path only contain portable activities, then the two services invoked by them are indirect dependent with each other.

The SBD Algorithm shows the process for component service selection. The input of the algorithm is the *DDG* of a process representing a service composition. The output of this algorithm is the set of component services chosen for the process and the site information at which each portable activity must be executed. For each directed sub-graph of the given *DDG,* pick an fixed activity *s* on random and choose a matched service for *s*. Then search this sub-graph

Algorithm2(SBD): select component services using data dependency graph
Input: $DDG(V, E_d)$ of a process
Output: component services chosen for the process, site information for each portable activity
begin
1 **for** *each Activity a* ∈ *V*
2 **do** *color[a]* ← *WHITE*
3 *a. Site* ← *NULL*
4 **end for**
5 **for** *each connected sub-graph in DDG*
6 **do** *choose a fixed activity s randomly in the sub-graph*
7 *color[s]* ← *GRAY*
8 *Q* ← ∅ //clear the Queue
9 *matched_instance* ← *findMatch(s)*
10 //get matched service instance for activity *s* at any site
11 *s. Site* ← *Service_instance. Site*
12 *ENQUEUE(Q, s)*
13 **While** *Q* ≠ {∅}
14 **do** *Activity u* ← *DNQUEUE(Q)*
15 **for each** *v*∈*Adj [u]*
16 **do if** *color[v] = WHITE* **then**
17 **if** *v is a fixed activity* **then**
18 *matched_instance* ← *findMatch (v, u.Site)*
19 //find a matched instance at the
20 // site where u is located at
21 **if** *matched_instance* ≠ *NULL* **then**
22 *v.Site* ← *u.Site*
23 **else** //there does not exist a matched
24 // instance at the site of u
25 *matched_instance* ← *findMatche(v)*
26 *v. Site* ← *matched_instance. Site*
27 **end if**
28 **else** //v is a portable activity
29 *v.Site* ← *u.Site*
30 **end if**
31 *color[v]* ← *Gray*
32 *ENQUEUE(Q, v)*
33 **end for**
34 *Color[u]* ← *BLACK*
35 **end while**
36 **end for**
end

using *s* as source node using colored path searching strategy without consideration the direction of edges. For the fixed activity *v* visited, choose a matched service instance from the site where *v*'s parent in the search tree is located at if there exist such instances. Otherwise choose one from other sites. The site information of chosen instance is recorded as the location where *v* is to be executed. For the portable activity *v* visited, use the site information of *v*'s parent as the location where *v* will be executed. This means the partition at which the portable activity *v* must be executed has also been determined. In this way, a portable activity is merged to another (fixed or portable) if there is a data dependency edge between them, economizing a message flow across partition. The procedure continues until all the activities in the sub-graph has been visited. The time complexity of SBD is linear in the size of *DDG*.

Since the randomness of this algorithm, running once could not guarantee the preferable solution. Consequently it must be run M times and choose the one that yield the least network traffic (the one with least number of data messages across different sites). According the experimental result, the amount of network traffic could be reduced obviously when M > |*V*|. To continue with the same example in Fig. 3, we give a possible distribution of service instances (See Fig. 5) needed by the example composite service. As Fig. 5 shows, there are two equivalent instances for each kind of service. These instances are deployed at four distinct nodes from *C1* to *C4*. A walk-through of the SBD is given in Fig. 6 and the result of it is shown in Fig. 5.

D. Process Decentralization

After component services are selected and portable activities are assigned in the last step, we convert composite service into decentralized partitions which re-implement the semantics of the centralized specification. Each subset of component services that are located at one site is associated by a partition consisted of fixed activities invoking them as well as portable activities determined to be executed at this site. One partition maintains the transitive dependencies

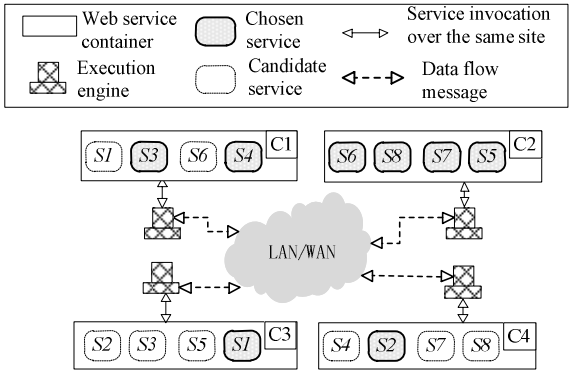


Figure 5. Distribution of service instances for the Illustrative example.

between the activities belong to it. These partitions are executed independently at distributed locations (collocated with their associated services) and pass data and control flow messages when necessary in a peer-to-peer fashion. The technique we use for process partitioning is the basic approach of decentralizing composite service which is similar to the work in [9].

III. PERFORMANCE EVALUATION

To evaluate the benefits of our component service selection algorithm, we perform a set of performance analysis tests where the SBD algorithm is evaluated. For comparison, we also implement a random algorithm which randomly selects a functionally qualified component service for each process activity that needs a service invocation.

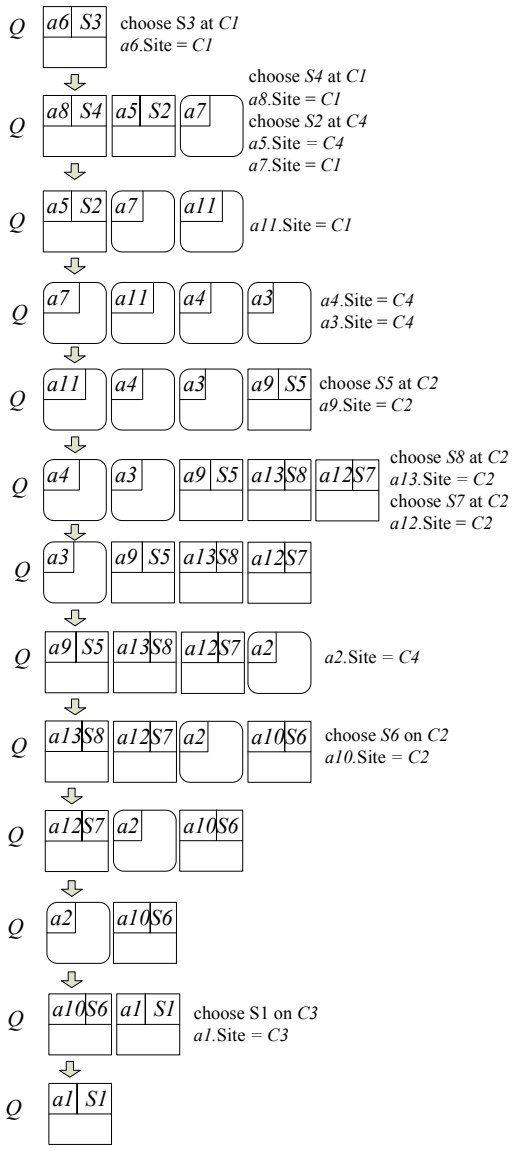


Figure 6. A walk through of SBD

System Design. The decentralized orchestration system used for testing consists of multiple execution engines running at distinct servers connected by LAN. We make use of our BPMN engine developed in [10], which support decentralized execution of composite service presented by BPMN partitions. Each BPMN engines is released as a web service and could be dropped into an Axis2 [29] container running on a Tomcat server. Component services are also deployed in these Axis2 containers installed at multiple servers. According to the architecture we discussed in the previous section, an engine is placed at each server collocated with a service container.

The process of executing a composite service is as follows. The centralized Service Bus create an instance for the needed composite service when receive a request. Then the Service Selection Module chooses component services and bind them for the BPMN specification. The service selection module could implement the SBD algorithm and random algorithm from our need. After component services are chosen, the instance is transformed to decentralized partitions by the Partitioning Module using the criteria discussed in section 3 (If random algorithm is used instead of SBD, then use the partitioning technique in [2] to place the portable activities). Then the BPMN Engines are invoked by Engine Invoking Module to send and execute the created partitions. Each engine has a thread pool and deals with tasks in a FIFO order. The result of composite service execution is sent by the partition which generates it to the Result Proxy module on the Service Bus. All the data are passed around using SOAP over HTTP.

Experimental Set Up. Our experimental setup for testing decentralized orchestration is as follows. We use six Intel Pentium based machine (1.86G CPU, 1GB RAM), each of which running a BPMN engine and an Axis2 container. The Service Bus with relative modules is installed on another machine with the same hardware resource.

Performance analysis. First, we take inspiration from the motivating example in Fig. 3 in order to verify the benefits of SBD where the number of equivalent instances for each kind of service and the distribution of service instances over network are varied.

We transform the example taverna workflow file into BPMN specification that could be executed by our engine. The example in Fig. 3 is actually a fragment of the Taverna workflow which consists of only beanshell processors [13], we transform some of the processors into script task [11] that could be interpreted by our engine (the relative java file needed are added in our engine), and the rest are released to eight component web services discussed in section 2.1.

We vary the instance number for each kind of service (represented by N_I) from 2 to 6. For reach value of N_I we generate a distribution of service instances on random but ensure the instances that have the same functionality are deployed at different containers. We computed the execution time for different decentralizations (determined by the SBD

versus random method), and varying size of input data size (between 100KB and 2000KB). For each combination of data size and N_I the experiment has been run independently 100 times. The execution time is calculated by taking the average elapsed time of 100 runs. We take the mean execution time of all combination of N_I by running the experiments iteratively on 2 to 10 instances for each kind of service.

Fig. 7 shows the results where the x-axis displays the input data size in Key Bytes and y-axis displays the average execution time in milliseconds. It confirms that the decentralized solution determined by the SBD algorithm result in a mean speedup ratio of 1.29 when N_I and distribution of service instances are varied. To explain the results, when using SBD algorithm the component services that have data dependence relation are placed at the same site as much as possible, which reduce the amount of network traffic than the random method since data exchanges over the same site do not generate network traffic. Only when the input data size does not exceed the capacity of the system (lower than 300KB) is the decrease of in execution time is not significant.

Second, in order to evaluate further effectiveness of the SBD algorithm on different composite services with varying number of activities and data operations, we implement a simulator (developed in JDK 1.6) which could generate random, acyclic processes and simulate the network traffic generated by their decentralized execution, i.e. the amount of the sizes of messages sent over the network. Each process generated has a random number of activities (range from 10 to 50). And the number of fixed activity is limited within the total number the process activity. The instance number for each kind of service is fixed at 10 and the number of service container is fixed at 20. For each service composition, we generate a set of def-use relations between activities on random, the number of which is linear to the size of the process. The size of data flow messages between component services is approximated by the input data size of the process. Furthermore, a possible distribution of service instances for each process is also generated randomly and with the constraints of container and instance number. We generated 1000 random processes, and for each of them, we implement the SBD algorithm and random service selection method. The size of input messages is varied between 500KB and 1000KB. Each measurement represents the average network traffic computed over the 1000 random composite services.

Fig. 8 shows the ratio of network traffic caused by decentralized orchestration determined by the SBD algorithm relative to the network traffic determined by random service selection method. The results confirm that the SBD algorithm is able to significantly reduce network traffic when compared with the random algorithm. The SBD algorithm caused about 73% on average of the network traffic due to the selection method on random during decentralized execution.

IV. RELATED WORK

Much work has been done in improving the performance of composite service execution. There two basic ways to achieve this [6]: One based on parallelizing components when dependencies permit, and the other based on decentralize the orchestration. The former is permitted both in centralized and decentralized execution while the latter is used to tackle the performance bottleneck caused by the centralized manner, in which all data exchange between component services must flow back and forth through the centralized coordinator and lead to unnecessary traffic on the network.

Approaches for decentralizing composite service have been proposed. Research [2] present first work in order to reduce the communication cost and maximizing throughput of composite service execution. Their contributions take on the technique to convert a composite service into an equivalent set of decentralized partitions. [6] is based on the same approach and focus on issues related to synchronization and coordination between decentralized partitions in order to

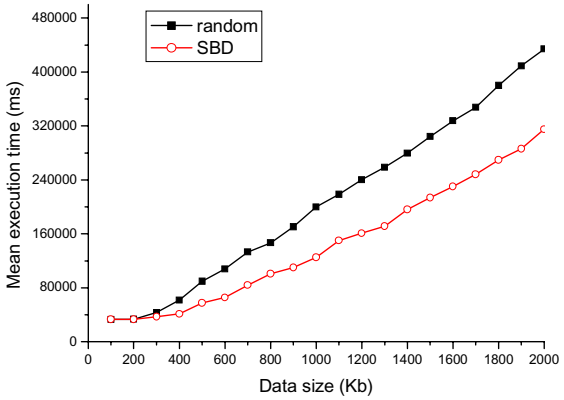


Figure 7. Mean execution time comparison between random and SBD algorithm

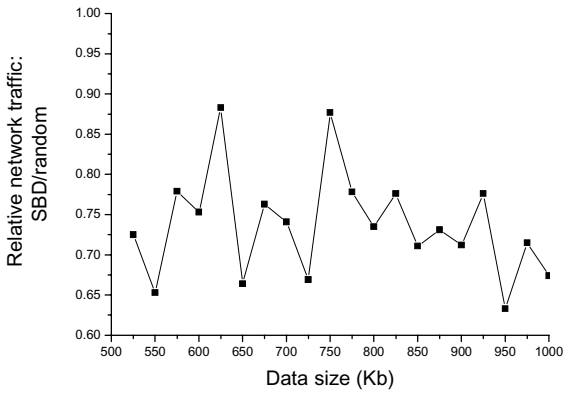


Figure 8. Relative network traffic: SBD/random (average computed over 1000 random, acyclic processes).

improve concurrency. Similar partitioning approaches have been applied to different needs such as error recovery and fault handling [7] with the goal of reducing response time. [5] investigate how different topologies generated by decentralized orchestration of component service are affected by variations in WAN conditions and present a system that adapt these changes. [3] and [4] present similar decentralization scheme to route traffic efficiently while [1] propose a hybrid architecture based on centralized control flow, decentralized data flow which maintains the robustness of centralized orchestration but facilitates the efficiency of decentralization.

In contrast to previous work, our technique is aim to reduce the network traffic of decentralized execution by a service selection method. It is based on the idea of choosing matched service instances that have data dependence relations from the same site, in order to make the data flow messages across different sites that generate network traffic is as less as possible.

V. CONCLUSION

In this paper, we introduce an approach to service selection with the goal of improving the performance for decentralized execution of composite services. First, the data dependence relationships between component services are computed. Second, an algorithm for service selection, SBD, is proposed based on the idea of selecting qualified instances from the same site for service components when they have data dependence relationships. By this means the data flow messages across different sites which generate network traffic are reduced during decentralized execution. The time complexity of SBD is linear in the size of the composite service. Our experiment results show significant performance improvements from SBD for the decentralized execution of our example composite service and 1000 simulated service compositions. For the same hardware resource, the decentralized orchestration determined by SBD performed better than the ones determined by the random method across a range of parameters for message sizes and number of service instances. The performance evaluation concludes that the decrease in network traffic by use of the SBD algorithm results in a reduction of approximately 30% in execution time on average.

REFERENCES

[1] A. Barker, J. Weissman, and J. van Hemert, Eliminating the middleman:Peer-to-peer dataflow. In 17th International Symposium on High-Performance Distributed Computing, pages 191–200, MA,USA, 2008.

[2] M. G. Nanda, S. Chandra, and V. Sarkar, Decentralizing execution of composite web services. In OOPSLA, pages 170–187, 2004.

[3] W. Binder, I. Constantinescu, and B. Faltings. Decentralized orchestration of composite web services. In IEEE International Conference on Web Services (ICWS'06), Chicago, IL, September 2006.

[4] B. Benatallah, Q. Sheng, and M. Dumas. The self-serv environment for web services composition. IEEE Intelligent Systems, 7(1): pages 40–48, 2003.

[5] G. Chafle, S. Chandra, N. Karnik, V. Mann and M. G. Nanda, Improving Performance of Composite Web Services Over a Wide Area Network. In IEEE Congress on Services, 2007.

[6] M. G. Nanda and N. M. Karnik. Synchronization Analysis for Decentralizing Composite Web Services. International Journal of Cooperative Information Systems, vol. 13, no.1, pages 9–119, 2004.

[7] G. Cha_e, S. Chandra, V. Mann, and M. G. Nanda. Decentralized Orchestration of Composite Web Services. In WWW'04, May 2004

[8] A. V. Aho, R. Sethi and J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1985, pages 624–625.

[9] W. Fdhila, U. Yildiz and C. Godart, A flexible approach for automatic process decentralization using dependency tables. In IEEE International Conference on Web Services, pages 847-855, 2009.

[10] Y. Ji, H. Sun, X. Liu, J. Zeng, S. Bai, A decentralized framework for executing composite services based on BPMN, In Service computation 2009.

[11] BPMI.org, OMG. Business Process Modeling Notation 1.1. http://www.omg.org/spec/BPMN/1.1

[12] http://www.myexperiment.org

[13] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver and K. Glover, M.R. Pocock, A. Wipat, and P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics, Oxford University Press, London, UK, 2004. Pages 3045–3054.

[14] http://ws.apache.org/axis2