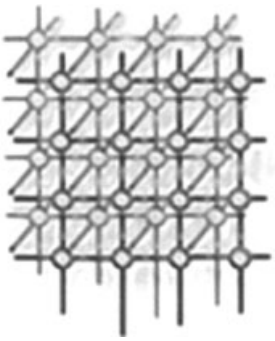
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **20**:361–376
Published online 5 July 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1218



A peak load control-based orchestration system for stable execution of hybrid services

Yonghwan Lee*,† and Dugki Min

School of Computer Science and Engineering, Konkuk University, Hwayang-dong, Kwangjin-gu, Seoul 133-701, Korea

SUMMARY

The prosperity of Internet results in a workflow executing engine's performance instability due to the request congestion for a short period of time. This paper proposes a peak load control (PLC) based orchestration system that can stably execute hybrid services. The PLC mechanism uses the delay time algorithm for controlling a BPEL engine's heavy peak load caused by the request congestion for a short period of time. In order to prove the stable performance of the PLC-based orchestration system, we analyze the proposed delay time algorithm. According to our experimental results, the proposed delay time algorithm can stably execute structured activities of WSBPEL specification in heavily overloaded state after the saturation phase and has an effect on controlling the states of peak load. In this paper, we also describe a hybrid service architecture model that can represent both Web Services and existing EAs as same type of services. Copyright © 2007 John Wiley & Sons, Ltd.

Received 19 March 2007; Accepted 1 April 2007

KEY WORDS: service composition; hybrid services; WSBPEL; peak load control; orchestration system

1. INTRODUCTION

The composition of service to form new, aggregate services is the domain of Web Services choreography [1–3]. Choreography languages, such as WSBPEL [4] and BPML [5], define an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to-business spaces [6]. However, Web Services-based approaches toward business process integration consider only Web Services as targets for integration [7,8]. Because there is no means to represent existing enterprise applications as a business partner of WSBPEL business process, existing enterprise applications cannot be represented in a business

*Correspondence to: Yonghwan Lee, School of Computer Science and Engineering, Konkuk University, Hwayang-dong, Kwangjin-gu, Seoul 133-701, Korea.
†E-mail: yhlee@konkuk.ac.kr

process when a business process is described by WSBPEL. Many existing enterprise applications that have been developed by RMI, EJB, CORBA, etc. are not taken into account for a WSBPEL-based service composition. Some mechanisms are required for representing both Web Service and existing enterprise applications as same type of services.

The prosperity of Internet results in instable performance due to the request congestion for a short period of time [9,10]. Moreover, the grid workflow systems generally require fixed-time constraints [11]. Accordingly, a workflow execution engine in grid environments also requires some mechanisms for automatically solving the temporary services congestion. The peak load control (PLC) is the mechanism for preventing thrashing in transaction processing system by controlling the number of concurrently running transactions [12]. The term thrashing generally describes a phenomenon where an increase of the load results in decrease of throughput (or another-related performance measure). Usually, we can distinguish a load-throughput function into three phases: underload, saturation, and overload. The underload phase is the state of light load with sufficient resources available and the throughput grows almost nearly to make use of possible parallelism in the system. The saturation phase is the state that reaches the highest throughput. When the finite capacity of the system becomes effective, the throughput function flattens out. After the saturation phase, further increasing load will not lead to an asymptotic approach to the saturation bound but sometimes will cause a sudden drop in throughput. In other word, the phenomenon of thrashing happens at overload phase. Generally speaking, at least the two following classes of factors make an effect on the overload: algorithmic overload (e.g. list operations, sorting, searching, etc.) and insufficient resource capacity [13]. Mutual impediments are also known to stem from contention for either physical resource (memory, processor) or logical resource (data granules). The former is usually called resource contention (RC), the latter is data contention (DC). Knowing that thrashing threatens in workflow execution engines in grid environments, we have to think about countermeasures to limit the load to prevent the system from overload.

In this paper, we describe the PLC-based orchestration system that can deploy and access the hybrid services represented by the proposed hybrid service architecture model. We use the delay time mechanism for the PLC. The mechanism calculates a delay time for each working thread at regular intervals. In order to calculate the delay time, the following factors are considered: over speed of transaction, the baseline delay, and slope of a download curve. In order to prove stable performance of the proposed mechanism, we apply the mechanism to the hybrid orchestration system. According to our experimental results, the proposed delay time algorithm can stably control the heavy overload after the saturation point and has an effect on controlling peak load.

This paper also proposes the hybrid service architecture model that can represent both Web Services and legacy services as partners in WSBPEL without the interoperability problem generated by extending WSDL. The hybrid architecture model uses two refinement phases (mapping phase, completion phase) for making the hybrid service architecture model from legacy architecture model, such as RMI architecture model, CORBA architecture model. During mapping phase, the same concepts are mapped between the legacy system architecture and the Web Service architecture. However, there are missing concepts that exist in the Web Service Architecture but do not exist in legacy architecture. During completion phase, similar or new concepts are made and mapped into the Web Service architecture for the missing concepts.

This paper is structured as follows. The next section presents the related work. Section 3 presents the proposed hybrid services architecture model. Section 4 describes the PLC-based hybrid service

DOI: 10.1002/cpe

orchestration system and the delay time-based PLC mechanism. Section 5 shows the experimental results of the mechanism. Finally, we conclude in the last section.

2. RELATED WORKS

In order to represent both Web Service and existing enterprise applications as same type of services, many recent WSBPEL vendors have provided Web Service-based orchestration system products, such as the Oracle BPEL Process Manager, the IBM WebService Business Integration Server Foundation, the ActiveBPEL Engine, and the Twister. However, some of them only support Web Service. Another does not use WSBPEL specification but its own specification for legacy service composition.

Currently, JOpera [14] and enterprise service bus (ESB) are proposed for legacy service composition. JOpera provides a rapid service composition tool offering a visual language and autonomic execution platform for building distributed applications out of reusable services, which include but are not strictly limited to Web services [15]. JOpera helps you to deal with heterogeneity. The ESB provides a new way to build and deploy enterprise service-oriented architectures (SOA). The ESB supports platform-neutral connection to any technology in the enterprise, e.g. Java, .Net, mainframes, and databases. The purpose of them is not for service composition but application integration. So, they generally focus in adapters to enable integration with a wide variety of enterprise applications. Therefore, they cannot represent legacy service as partners in WSBPEL. Web Services invocation framework (WSIF) is a simple Java API for invoking Web Services, no matter how or where the services are provided. WSIF enables developers to interact with abstract representations of Web Services through their WSDL descriptions instead of working directly with simple object access protocol (SOAP) APIs, which is the usual programming model. With WSIF, developers can work with the same programming model regardless of how the Web Service is implemented and accessed. However, WSIF needs to extend WSDL of each legacy service in order to adapt the legacy services like RMI, EJB, and JMS. The extension of WSDL brings the interoperability problem to other applications except WSIF. So, we describe legacy services as another description format like RMI service description (RMISD).

In order to prevent thrashing in overload phase, the fixed upper bound of the maximum number of transaction, analytical models [14,16,17], adaptive load control mechanism [18], and concurrent programming [19–21] have been suggested. Several solutions also are compared [17]: do nothing, fixed upper bound, theoretically derived 'rule of thumb'. The first solution relies on self-regulating market mechanism. If the service (throughput, response time) becomes worse, fewer people want to use it. The second solution limits the maximum number of concurrent transactions. However, when the transaction load is constant and the value is chosen appropriately, this solution may work. The third solution uses analytical models which suggest some conditions that must be satisfied to prevent thrashing. Tay *et al.* [14], for example, claim that K^2n/D should be less than 1.5 where K is the number of data items accessed by each transaction, n is the concurrency level, and D is the database size. Iyer [16] suggests then the mean number of conflicts per transactions should not exceed 0.75.

In contrast with existing load control mechanisms, our approach does not control the number of concurrent active threads but the delay time of each active thread for solving thrashing phenomenon.

DOI: 10.1002/cpe

3. HYBRID SERVICE ARCHITECTURE MODEL

In this section, we propose the hybrid service architecture model that represents legacy services, such as RMI objects and EJB components, as a partner in WSBPEL. The hybrid service architecture is the re-constructed architecture made by refining legacy system architectures in view of the Web Service architecture. The Web Service architecture can be categorized into two kinds of conceptual model [22]: service model and message model. The service model describes main concepts in service-oriented view. A service has one or more service descriptions. A service description has one or more service addresses and service interfaces for a service. A service implements the service interfaces and service requestor uses the services provided by service providers. A service interface has one or more operations. The message model represents messages interacting between services. An operation has zero or more messages and a message has both a message name and one or more message part. The message part is used for overriding methods. A message part refers to a message type.

In order to build the hybrid service architecture from a specific legacy architecture model, two refinement phases are needed: mapping phase and completion phase. During the mapping phase, the same concepts are mapped between the legacy system architecture and the Web Service architecture. However, there are missing concepts that exist in the Web Service Architecture but do not exist in the legacy architecture. During the completion phase, similar or new concepts are made and mapped into the Web Service architecture for the missing concepts. Figure 1 is the conceptual mapping between the Web Service and Java RMI architecture during two phases.

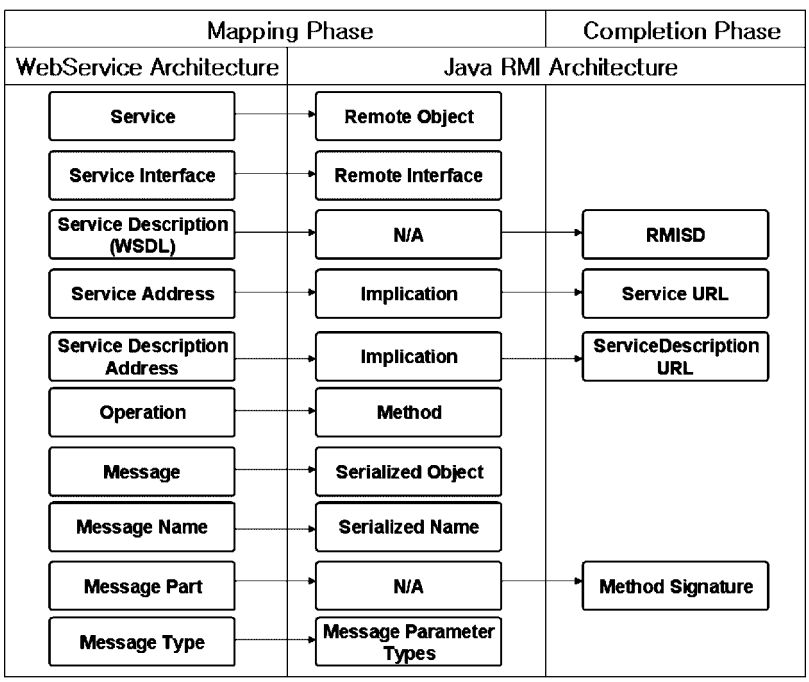


Figure 1. Conceptual mapping during two phases.

DOI: 10.1002/cpe

In order to represent RMI objects as Web Services in WSA and WSDL, the RMI architecture model needs to be refined in view of the Web Service architecture model. For this refinement, the conceptual mapping between two architectures is needed. A remote object, remote interface, method, method name, method parameter types of the Java RMI architecture are mapped to a service, service interface, message, message name, and message type of the Web Service architecture, respectively. However, the concepts of the service description and message part in the hybrid service architecture do not exist in the Java RMI architecture. The service address is the URL for accessing a Web Service and the service description address is the URL for accessing a WSDL document. In the Java RMI architecture, the service address and service description address are not explicitly described. In order to map the Java RMI architecture to the Web Service architecture, the message part and service description need to be redefined and the service address and service description address also need to be explicitly specified in hybrid service architecture during the completion phase.

During the completion phase, the RMISD, ServiceURL, ServiceDescriptionURL, Method Signature of the Java RMI architecture are mapped to a WSDL, service address, WSDL address, and Message Part of the Web Service architecture, respectively. The role of message part in the Web Service architecture is generally similar to the role of overriding methods in the Java RMI architecture. In other words, the message part is used to differentiate the overridden methods in the Java RMI architecture. So, the format of the method signature is used for the role of message parts. The method signature format is as follows: 'return type; method name; (parameter types); (variable names)'. In Java RMI architecture, because such a service description like the WSDL in the Web Service architecture does not exist, we create the RMISD document. The RMISD describes information about RMI-based services. Let us assume *shippingPT* RMI remote interfaces of Figure 2 for easy description of the message part and RMISD. According to the previous method signature format, the method signature of the *shippingPT* interface is the following: 'String;requestShipping;(String);(shipping)'. The *ServiceInterfaceName* element describes the name of the *shippingPT* Java RMI remote interface. The *MethodSignature* element also describes the *requestShipping* method signature. The *ServiceDescriptionURL* element has an URL for finding a RMISD of the *shippingPT* RMI service and the *ServiceURL* is used for accessing the *shippingPT* RMI service. The RMISD file extension is used to represent that the service is not implemented by Web Services but Java RMI objects.

After mapping the Java RMI architecture to the Web Service architecture, a Java RMI object can be represented as a service in WSBPEL business process. The service representation in WSBPEL business process can be categorized into two following methods: WSDL-based service representation and RMISD-based service representation. Figure 3 shows the difference between the WSDL-based service representation and the RMISD-based service representation.

The WSDL-based service representation represents a Java RMI object as a service in WSDL document. The RMISD-based service representation represents a Java RMI object as a service in a RMISD document. A WSBPEL process represents all partners and interactions with these partners in terms of abstract WSDL interface (i.e. *portTypes* and *operations*). In the WSDL-based service representation, the *partnerLink* of the WSBPEL document refers the *Role* in the *PartnerLinkType* of the WSDL document. In the WSDL document, an URL of a XML namespace definition is used for referring the service interfaces of RMISD. The *Role* of the *PartnerLinkType* refers to the *ServiceInterfaceNames* of another RMISD document.

Figure 4 shows the WSDL-based service representation.

DOI: 10.1002/cpe

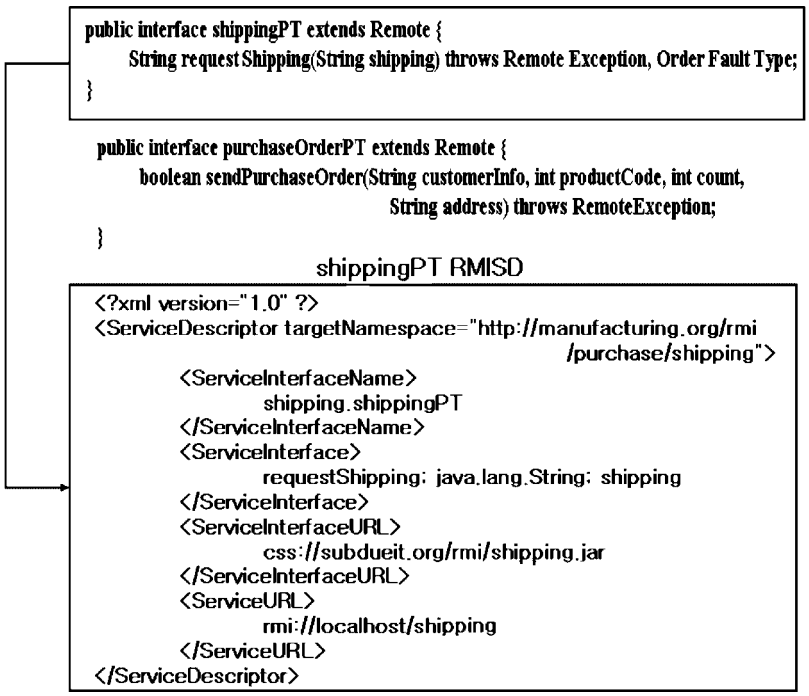


Figure 2. RMISD example for shippingPT Java RMI interface.

Two services are represented by two *PartnerLinkType* elements: the *purchasingLT* and *shippingLT*. The *purchasingLT* is a Web service and the *shippingLT* is a Java RMI object. In case of the *purchasingLT*, the description URL is identified by the namespace attribute in the definition element (i.e. http://manufacturing.org/wsdl/purchase.wsdl) and the name of service interface is identified by the name attribute (i.e. pos: purchaseOrderPT) of the *portType* child element of the *partnerLinkType* in WSDL. In case of the *shippingLT*, the description URL of a shipping service is identified by the namespace attribute in the definition element (i.e. rmi://manufacturing.org/rmi/ship.rmisd) and the name of service interface is identified by the name attribute (i.e. sos:shipping.ShippingOrder) of the *portType* of the *partnerLinkType* in WSDL.

In contrast to the WSDL-based service representation, in the RMISD-based service representation, the roles of the *PartnerLinkeType* refer to the *ServiceInterfaceNames* of the same RMISD document. Figure 5 shows the WSDL-based service representation. Three services are represented by three *PartnerLinkType* elements: the *purchasingPT*, *shippingPT*, and *shippingRequstor*. The *purchasingPT* RMI object is described as a service in this RMISD document. The *shippingPT* and *shippingRequstor* RMI objects are described as services in another RMISD document. So, the URL of namespace definition refers to RMISD files. The *PartnerLinkType* of the *shippingPT* has two following role elements: the *ShippingService* and the *ShippingRequestor*. The role name of the *ShippingService* has the *shipping:shippingPT portType*. The *portType* identifies the

DOI: 10.1002/cpe

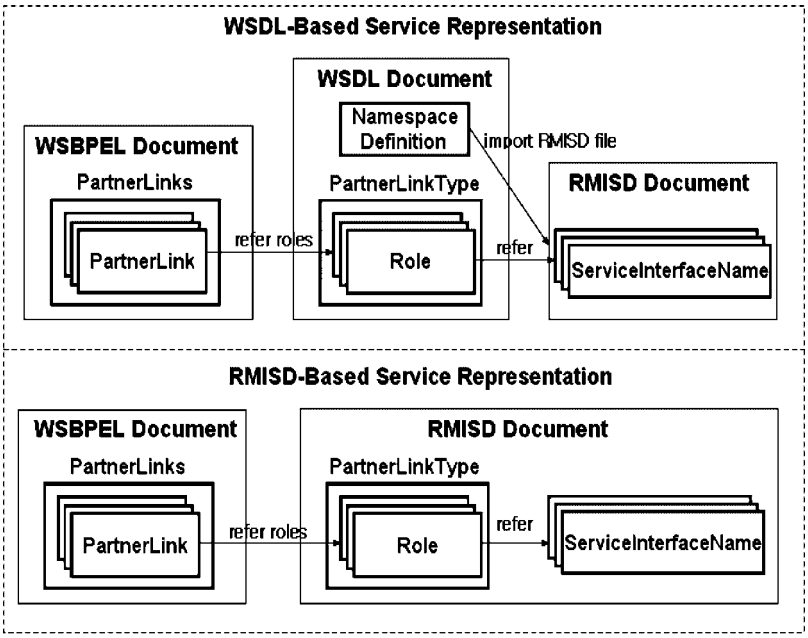


Figure 3. Two service representations in WSBPEL business process.

```
<?xml version="1.0"?>
<definitions name="SyncHelloWorld4"
        xmlns:pos="http://manufacturing.org/wsdl/purchase.wsdl"
        xmlns:sos="rmi://manufacturing.org/rmi/ship.rmisd"
        ...>
        <!-- omitted -->
  <plnk:partnerLinkType name="purchasingLT">
    <plnk:role name="purchaseService">
      <plnk:portType name="pos:purchaseOrderPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
  <plnk:partnerLinkType name="shippingLT">
    <plnk:role name="shippingService">
      <plnk:portType name="sos:ShippingOrder"/>
    </plnk:role>
  </plnk:partnerLinkType>

</definitions>
```

Figure 4. WSDL-based service representation.

shipping service described by the ship.rmisd document. The *shipping.shippingCallbackPT portType* identifies the purchase order service described in the purchaseOrder.rmisd. The *portType* name in the *partnerLinkType* needs to be equal to the value of the *ServiceInterfaceName* of the RMISD document.

DOI: 10.1002/cpe

```
<? XML version="1.0" ?>
<Servicedecriptor xmlns:plinke="http://manufacturing.org/purchseOrderLT
                  xmlns:sos="rmi://manufacturing.org/rmi/ship.rmisd"
                  xmlns:pks="rmi://manufacturing.org/rmi/shipCallBack.rmish>
  <ServiceInterfaceName>
    plinke.purchaseOrderPT
 </ServiceInterfaceName>
  <ServiceInterface>
          sendPurchaseOrder; java.lang.String, int, int, java lang,String; customeInfo,
                                        productCode, count,address
  </ServiceInterface>
  <ServiceInterfaceURL>
    rmi://localhost/purchaseOrderLT.jar
  </ServiceInterfaceURL>
  <ServiceURL>
    rmi://localhost/purchaseOrder
  </ServiceURL>
  <plink: partnerLinkType name="purchasePT">
    <plink:role name="purchaseService">
      <plink:portType name="plinke.purchaseOrderPT">
    </plink:role>
  </plinke: partnerLinkType>
  <plink: partnerLinkType name="shippingPT">
    <plink:role name="shippingService">
      <plink:portType name="sos.shippingPT">
    </plink:role>
    <plink:role name="shippingRequestor">
      <plink:portType name="pks.shippingCallbackPT">
    </plink:role>
  </plinke: partnerLinkType>
</Servicedecriptor>
```

Figure 5. RMISD-based service representation.

4. PLC-BASED ORCHESTRATION SYSTEM

In hybrid service architecture model, we propose the remodeling method for representing both Web Services and legacy services as partners in WSBPEL processes. In this chapter, we present the PLC-based hybrid service orchestration system that can stably execute hybrid services by the proposed hybrid service architecture model. Figure 6 is the software architecture of the orchestration system. The orchestration system is composed of the following main modules: the BPEL Engine, Hybrid Service Toolkit, Context Sharing Server, and Admin Console. The *BPEL Engine* is a workflow engine for executing business processes described in WSBPEL. Service providers use the *Hybrid Service Toolkit* for creating and deploying hybrid services to a service container. Service consumers use the *Hybrid Service Toolkit* for invoking the hybrid services. The *Context Sharing Server* is responsible for managing the state of the business process instance and performing authentication and transaction. The state of context sharing server is also shared by many hybrid orchestration systems. The *Admin Console* manages the hybrid orchestration system and the *Context Sharing Servers*. The *Hybrid Service Toolkits* supports the proposed hybrid service architecture.

Figure 7 is a class diagram of the *Hybrid Service Toolkit*. The *HSTKFactory* generates three following factory objects: *ServiceDescriptionReader*, *ServiceGenerator*, *and ServiceDeployer*. The three factory objects are created for legacy services, such as RMI, EJB, and CORBA. The three factory objects are similar to the adapters of the *Hybrid Service Toolkit* in Figure 7. In other words, clients invoke the hybrid services which are generated and deployed by the *Hybrid Service Toolkit*. The hybrid services cause the *BPEL Engine* to execute the business processes of WSBPEL. The

DOI: 10.1002/cpe

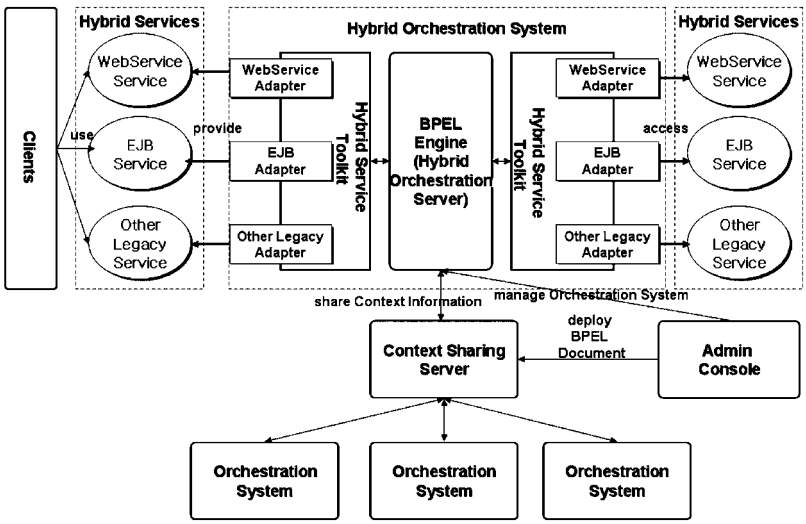


Figure 6. Software architecture of the orchestration system.

BPEL Engine uses the *Hybrid Service Toolkit* for accessing hybrid services. The *ServiceDescriptionReader* reads the hybrid service description remodeled by the hybrid service architecture and generates the *ServiceDescription* object. The *ServiceGenerator* generates a *Service* object using the *ServiceDescription parameter* object. A service consumer can consistently use hybrid services by invoking the hook method of the *Service* object. The *ServiceDeployer* generates the hybrid services described in the *ServiceDescription* and deploys the hybrid services into a service container. The *ServiceDeployer* uses the *Callback* object to handle input parameters when deploying hybrid services.

Figure 8 is the software architecture of the BPEL Engine. The *Process Manager* executes WS-BPEL processes and supports many kinds of activities described by WSBPEL standard specification. The *Alarm Service* is responsible for executing processes at the described time. The *Lock Service* shares the information of process locking among the *Context Sharing Servers*. The *Message Service* receives messages through the hybrid service toolkit from clients and generates the *Caller* object for accessing hybrid services. The *Worker Service* manages many threads used by the orchestration system.

The *Admin Console* uses the *Remote Service* for accessing the *BPEL Engine*. The *Remote Service* is the boundary service for controlling the *BPEL Engine* from the *Admin Console*. The *CSS Service* is responsible for communicating with the *Context Sharing Server*. The *Context Sharing Server* is responsible for managing the consistent state of business processes in multiple orchestration system environments. In view of the qualities of software architecture, the *BPEL Engine* is the scalable and reliable system using the *Context Sharing Server*.

In WSBPEL specification, there are many structured activities, such as sequence, switch, while, flow, and pick. The structured activities prescribe the order in which a collection of activities take place and can be nested and combined in arbitrary ways. The hybrid service orchestration system

DOI: 10.1002/cpe

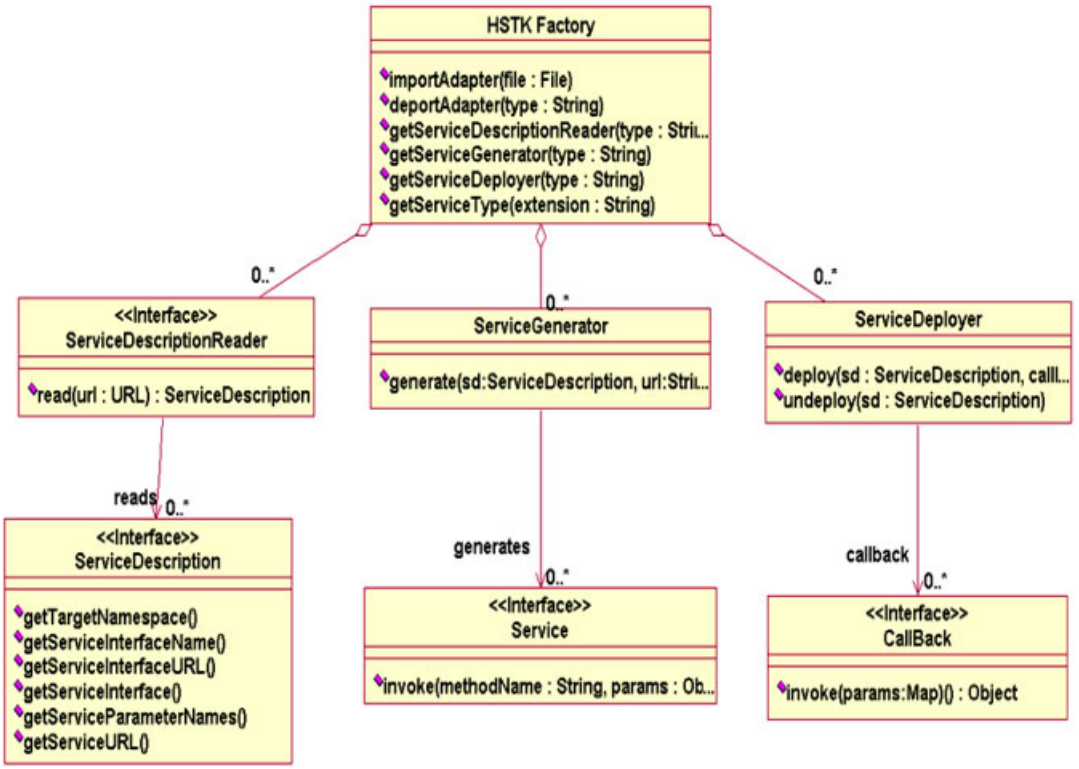


Figure 7. Design of the hybrid service toolkit.

uses Worker threads for executing those complex activities. The *Worker Service* of Figure 8 manages the *Worker threads*.

Figure 9 shows how to use the *Worker threads* and how to apply them in order to execute structured activities of WSBPEL specification. The *Worker Service* gets a Worker thread from the *Worker Thread Pool*. The *Worker thread* gets delay time from the *WorkerManager*. After the *Worker thread* sleeps for the delay time calculated by the *WorkerManager*, the *Worker thread* executes structured activities sequentially or concurrently. Finally, the *Worker* increments the number of transaction processed by the *Worker thread*.

Figure 10 is the pseudocode for the WorkerManager's delay time algorithm. After sleeping during interval time, the *WorkerManager* gets the number of transactions processed by all *Worker threads* and the maximum transaction processing speed configured by a system administrator. And then, the *WorkerManager* calculates the transaction per milliseconds (TPMS) by dividing the number of transactions by the maximum transaction processing speed and calculate the over speed between the TPMS and the maximum transaction processing speed. If the value of the over speed is greater than zero, the system is considered as an overload state. Accordingly, it is necessary to control the overload state. On the contrary, if the value of the over speed is zero or less than zero, it is not necessary to control the transaction processing speed.

DOI: 10.1002/cpe

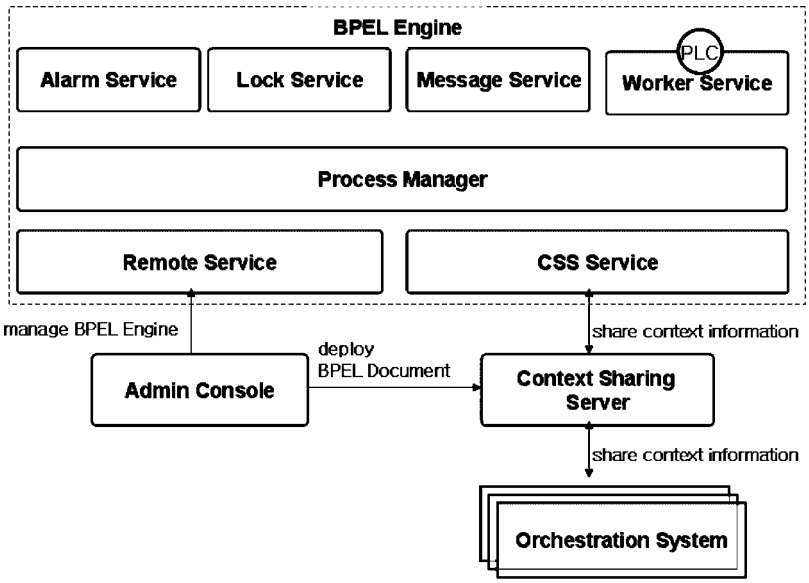


Figure 8. Software architecture of the BPEL engine.

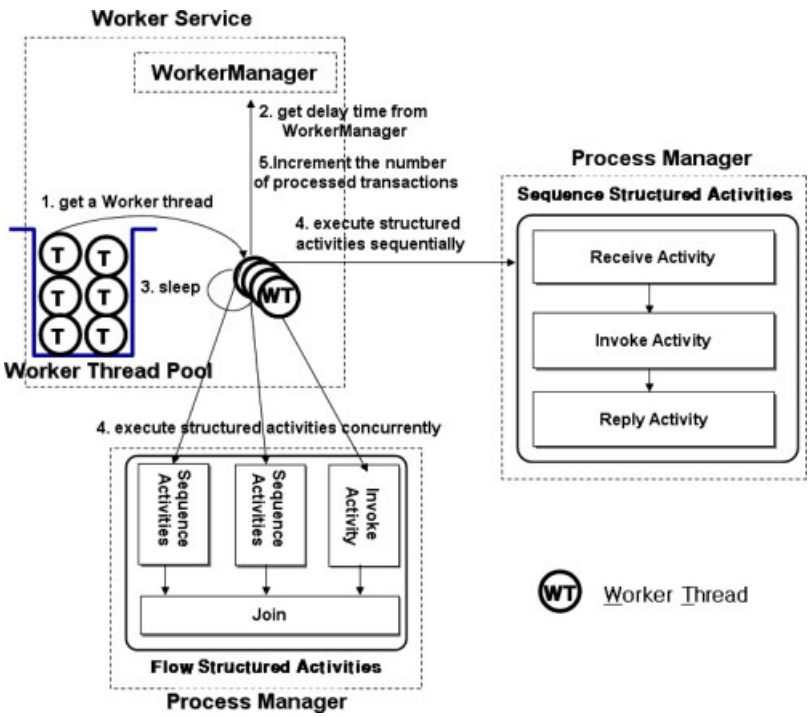


Figure 9. Worker threads for executing complex structured activities.

DOI: 10.1002/cpe

```
1.while run_flag equals "true" do
2.  get interval time for checking load
3.  sleep for the interval time
4.  get the number of transactions processed during the interval time
5.  get the configured maximum speed
6.  TPMS := number of transactions / interval time
7.  over speed := TPMS – maximum speed
8.  If over speed >0 then
     8.1 get the previous delay time
     8.2 if previous delay time = 0
      8.2.1 previous delay time := 1
     8.3 get number of active worker thread
     8.4 new delay time:= over speed / number of active worker * previous delay time
9   else
     9.1 get current delay
     9.2 if current delay > δ
       9.2.1 new delay time := current delay * β
     9.3 else
       9.3.1 new delay time := 0
     9.4 end if
10. end if
11.end while
```

Figure 10. Pseudo code for WorkerManager's delay time algorithm.

For controlling the overload state, this paper uses the delay time algorithm of the *WorkerManager*. $\mathrm{OS}(t_{i+1})$ is the over speed between the transaction processing speed ($\mathrm{TPMS}(t_{i+1})$) at the time t_{i+1} and the configured maximum transaction processing speed (MTPS). The $\mathrm{OS}(t_{i+1})$ is calculated by applying formula (1). If the over speed $\mathrm{OS}(t_{i+1})$ is greater than zero, formula (2) is used for getting a new delay time $D(t_{i+1})$ at the time t_{i+1}. The $N(t_{i+1})$ of formula (2) means the number of active *Worker threads* at the time t_{i+1} and $D(t_i)$ means the delay time at the time t_i. If the $D(t_i)$ is zero, $D(t_i)$ must be set one:

$$\mathrm{OS}(t_{i+1}) = \mathrm{TPMS}(t_{i+1}) - \mathrm{MTPS} \tag{1}$$

$$D(t_{i+1}) = \mathrm{OS}(t_{i+1})/N(t_{i+1}) * D(t_i) \tag{2}$$

If the $\mathrm{OS}(t_{i+1})$ of formula (1) is zero or less than zero, a new delay time $D(t_{i+1})$ at the time t_{i+1} is differently calculated according to the delay time $D(t_i)$ at the time t_i. If the $D(t_i)$ at the time t_i is greater than the baseline delay δ, the $D(t_{i+1})$ is calculated by applying formula (3). On the contrary, if the $D(t_i)$ is same or less than the baseline delay, $D(t_{i+1})$ is set zero. The baseline delay is used for preventing repetitive generation of the over speed generated by suddenly dropping the next delay time in previous heavy load state. When the system state is continuously in state of heavy load for a short period of time, it tends to regenerate the over speed to suddenly increment the delay time at the time t_i and then suddenly decrement the delay time zero at the time t_{i+1}.

The baseline delay is the previous delay time at the time t_i that can decide whether next delay time at the time t_i is directly set zero or not. The β percent of formula (3) decides the slope of a

DOI: 10.1002/cpe

downward curve. However, if the delay time at the time t_i is lower than the baseline delay. The new delay time at the time t_{i+1} is set zero. Accordingly, when a system state changes from the heavy overload at the time t_i to the underload at the time t_{i+1}, The gradual decrement by the β percent prevents the generation of repetitive over speed caused by abrupt decrement of the next delay time:

$$D(t_{i+1}) = D(t_i) * \beta \tag{3}$$

$$D(t_{i+1}) = 0 \tag{4}$$

5. EXPERIMENTAL RESULTS

In order to prove performance stability of the PLC-based hybrid service orchestration system, we analyze the delay time algorithm of the *WorkerManager*. As for load generation, the LoadRunner 8.0 tool is employed. Transactions per second (TPS) is used as a metric for performance analysis. As the purpose of performance experiment is not to compare with other system but performance stability, we use a RMI-based hybrid service which inquires an emp table of oracle 9i database. The three following activities are executed in sequence: receive activity, invoke activity, and reply activity. The invoke activity invokes the RMI-based hybrid service. The maximum speed, δ and β for delay time algorithm are configured 388, 100, and 0.75 ms, respectively. The data size received from the emp table is 475 bytes. The number of concurrent users is 300 and the value of think time is 0. Maximum TPS of the proposed integration system is 327 TPS and average TPS is 325.

Figure 11 is a comparison of stability of performance between the orchestration system without the PLC and the proposed PLC-based orchestration system. The orchestration system without the PLC does not use the PLC mechanism but the proposed system uses the mechanism. Until concurrent user 220, two systems are similar in view of TPS and the maximum TPS of both systems is 327. However, as a number of concurrent users are more than 220 users, both systems

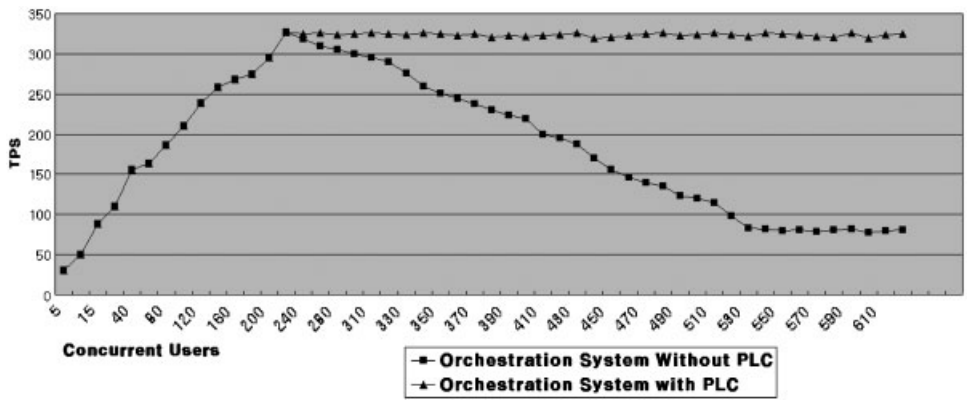


Figure 11. Comparison of performance stability.

DOI: 10.1002/cpe

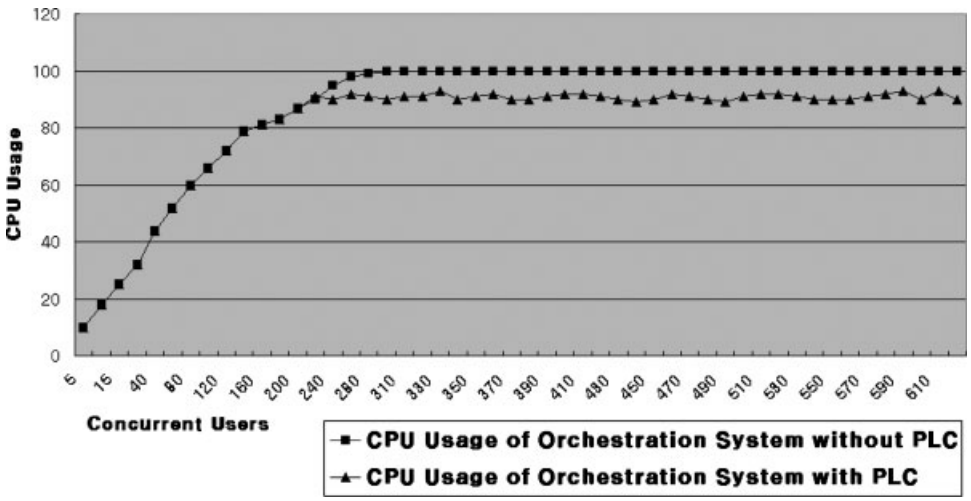


Figure 12. Comparison of CPU usage.

show different symptoms. The proposed PLC-based system holds 325 TPS on the average due to the PLC mechanism. However, the TPS of the orchestration system without the PLC goes down until concurrent users reach 520. As the concurrent users go over 520, the TPS of the orchestration system without the PLC holds 80.

Figure 12 explains the reason of different symptoms between the orchestration system without the PLC and the proposed PLC-based orchestration system. As the number of concurrent users is more than 280 users, the Non PLC-based system is in heavily overloaded state as to reach 100% CPU usage. However, because the PLC-based system has the PLC mechanism, the proposed system holds 92% CPU usage. The PLC-based system adds PLC mechanism to the hybrid orchestration system so that it can prevent the performance instability caused by the request congestion.

Figure 13 shows the relationship between the over speed and the delay time after the saturation point. This experimental result proves that the proposed delay time algorithm of the *WorkerManager* has an effect on controlling the over speed. As the number of concurrent users is more than 220 users, the over speed frequently happens. Whenever the over speed happens, each *Worker thread* sleeps for the delay time calculated by the *WorkerManager*. As the higher over speed happens, each *Worker thread* sleeps for the more time so that the over speed steeply goes down. Although the over speed steeply goes down, the delay time does not steeply go down due to the baseline delay value δ. As the baseline delay value is set 100 ms in this experiment, the delay time gradually goes down until the 100 ms. As soon as the delay time passes 100 ms, the next delay time is directly set zero.

Figure 13 shows that the over speed does not happen until zero delay time due to the slope of a downward curve. However, as soon as the delay time passes zero, the over speed again happens and the next delay time controls the over speed. Although the heavy request congestion happens in a BPEL Engine, the delay time-based PLC mechanism can prevent the thrashing state in overload phase and help a BPEL Engine to execute stably the complex structured activities of WSBPEL specification.

DOI: 10.1002/cpe

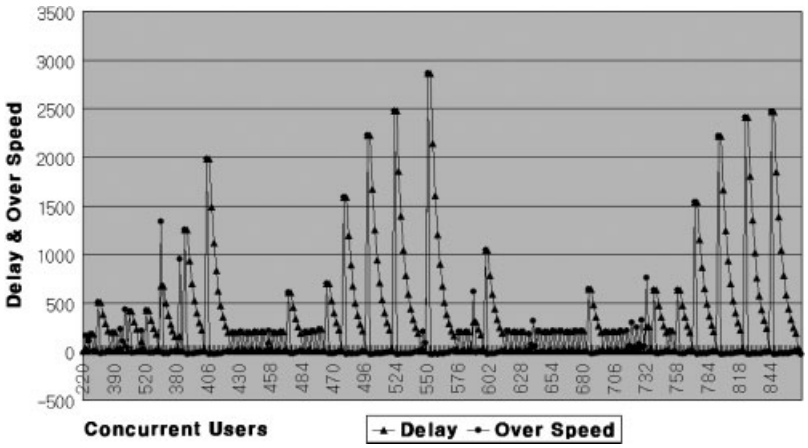


Figure 13. Over speed control by the delay time algorithm.

6. FUTURE WORKS AND CONCLUSIONS

In this paper, we provide the hybrid service architecture model that can represent both Web Services and existing EAs (i.e. legacy system) as partners in WSBPEL. This proposed hybrid service architecture model can represent both Web Services and EA as same services in WSBPEL. As an example, we show how Java RMI objects can be represented as services. In this paper, we also describe the hybrid service orchestration system for supporting the hybrid service architecture model. The orchestration system is composed of the BPEL Engine, the Hybrid Service Toolkit and the Context Sharing Server. The hybrid service toolkit helps service providers to provide the hybrid services and service clients to access hybrid services.

Moreover, in order to prevent the BPEL Engine's thrashing caused by request congestion for a short period of time, we provide the delay time-based PLC mechanism for solving RCs. In order to calculate the delay time, the mechanism considers three following factors for calculating the next delay time: over speed of transaction, baseline delay, and slope of a download curve. We also apply the mechanism to the hybrid orchestration system in order to prove stable performance of the proposed mechanism. According to our experimental results, the proposed delay time algorithm can stably control the heavy overload after the saturation point and has an effect on controlling peak load. In future works, we will research the dynamically changing optimal values, such as maximum speed, baseline delay value δ, slope of a download curve β.

REFERENCES

1. Ruoyan Z, Arpinar B, Aleman-Meza B. Automatic composition of semantic web services. *The 2003 International Conference on Web Services* (*ICWS'03*), http://lsdis.cs.uga.edu/lib/download/composition_short_icws.doc [June 2003].
2. Pajunen L, Korhonen J, Puustjarvi J. Adaptive web transactions: An approach for achieving the atomicity of composed web services. *Proceedings of EuroWeb Conference*, 2002.

DOI: 10.1002/cpe

3. Arkin A, Askary S, Fordin S, Jekeli W, Kawaguchi K, Orchard D, Pogliani S, Riemer K, Struble S, Takaci-Nagy P, Trickovic I, Zimek S. *Web Service Choreography Interface* (*WSCI*) *1.0*. Published on the World Wide Web by BEA Systems, Intalio, SAP, and Sun Microsystems, 2002.
4. Curbera F, Goland Y, Klein J, Leymann F, Roller D, Thatte S, Weerawarana S. *Business Process Execution Language for Web Service* (*BPEL4WS*) *1.0*. Published on the World Wide Web by BEA Corp, IBM Corp and Microsoft Corp, August 2002.
5. Shapiro R. A comparison of XPDL, BPML, and BPEL4WS. *Draft Document*, Cape Visions, http://xml.coverpages.org/Shapiro-XPDL.pdf [May 2002].
6. Liu JX, Zhang SS, Hu JM. A case study of an inter-enterprise workflow-supported supply chain management system. *Information and Management* 2005; **42**(3):441–454.
7. Thatte S. *XLANG*. Published on the World Wide Web by Microsoft Corporation, 2001.
8. Weerawarana S, Curbera F, Matthew J, Duftler, Epstein DA, Kesselman J. Bean markup language: A composition language for JavaBeans components. *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, USENIX, January 2001.
9. Floyd S, Handey M, Padhyte J, Widmer J. Equation based congestion control for unicast applications. *Proceedings of ACM SlGCOMM' 2000*, May 2000; 43–56.
10. Mahdavi J, Floyd S. TCP-friendly unicast rate-based flow control. *Technical Note* sent to the end2end-interest mailing list, January 1997.
11. Chen J, Yang Y. Multiple states based temporal consistency for dynamic verification of fixed-time constraints in grid workflow systems. *Concurrency and Computation*: *Practice and Experience* 2007; **19**(7):965–982.
12. Denning PJ. Thrashing: Its causes and prevention. *Proceedings of the AFlPS FJCC* 1968; **33**:915–922.
13. Agrawal R, Carey MJ, Livny M. Concurrency control performance modeling: Alternatives and implications. *ACM TODS* 1987; **12**(4):609–654.
14. Tay YC, Goodman N, Suri R. Locking performance with dynamic locking. *ACM TODS* 1985; **10**(4):415–462.
15. Pautasso C, Alonso G. JOpera: A toolkit for efficient visual composition of web services. *International Journal of Electronic Commerce* 2004/2005; **9**(2):104–141.
16. Iyer BR. Limits in transaction throughput—why big is better. *IBM Research Report No. RJ6584*, IBM Research Division, Yorktown Heights, NY, 1988.
17. Sevcik KC. Comparison of concurrency control methods using analytic models. In *Information Processing*, vol. 83, Mason REA (ed.). North-Holland: Amsterdam, 1983; 847–858.
18. Heiss HU, Wanger R. Adaptive load control in transaction processing systems. *Proceedings of the 17th International Conference on Very Large Data Bases*, September 1991.
19. Douglas CS, Michael S, Hans R, Frank B. *Pattern-oriented Software Architecture*: *Concurrent and Networked Objects*. Wiley: New York, 2000.
20. Lavender RG, Schmidt DC. Active object: An object behavioral pattern for concurrent programming. *Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs*, Monticello, IL, September 1995; 1–7.
21. Doug L. *Concurrent Programming in Java* (2nd edn). Addison-Wesley: Reading, MA, 1999.
22. Austin D *et al*. Web services architecture requirements. *W3C Working Draft*, www.w3.org/TR/wsa-reqs [November 2002].

DOI: 10.1002/cpe