SED4BPEL: A Staged Event-Driven Architecture for High-Concurrency BPEL Engine

Dou Sun, Yongwang Zhao, Hao Zeng, Dianfu Ma
Institute of Advanced Computing Technology
National Lab of Software Development Environment
School of Computer Science and Engineering, Beihang University
No 37, Xueyuan Road, Haidian District, Beijing, China
{sunou, zhaoyw, zenghao}@act.buaa.edu.cn dfma@buaa.edu.cn

Abstract—Current BPEL engine products are difficult to meet the highly concurrent demands of increasing mission-critical business processes application. We follow the ideas of SEDA and propose a new architecture for high-concurrency BPEL engine, which we call SEDA4BPEL. In SEDA4BPEL, the implementation of BPEL related web services protocols is encapsulated into four primary event-driven stages, to provide independence, isolation and modularity. We also introduce two controllers to manage excessive concurrent process instances. We present the SEDA4BPEL design and the implementation of a BEPL engine based on this architecture. The evaluation results show that SEDA4BPEL applications exhibit high performance and robustness when handling massive concurrency.

Keywords—SOA; Web Services; BPEL; SEDA; High-concurrency;

I. INTRODUCTION

The popularization of Service-Oriented Architecture (SOA) in recent years has changed the application, system and software perspectives in many modern organizations [1]. Within SOA, a basic component is implemented as a service, which exposes its functionality in a platform-independent manner and can be discovered, orchestrated and consumed by clients or other services [2]. A more complex service can be made in form of process by services orchestration, and services interactions are described as activities in the process. The Web Services Business Process Execution Language (WS-BPEL, BPEL for short) [3], which was approved as an OASIS standard in 2007, has been a de-facto industrial standard for specifying and executing services orchestration based on business processes.

There is an urgent demand for constructing mission-critical business processes [4] by BPEL. Those processes are directly related to the core functions of organization and need to deal with massive simultaneous access. The credit and loan application of BankComm [5] is a typical example. This system includes eight major functional modules supported by more than 30 mission-critical business processes, involves 2700 outlets in 86 large and medium cities of China. There are 73 process definitions and the longest process path is 52 steps. 1800 process instances and 21,000 task instances are created daily at maximum. The number of daily online staff is 800 at average and 1,400 at maximum (the data was provided by CVIC SE [6], one of the biggest software corporations in China). This highly concurrent requirement presents a big challenge to the system design of BPEL engine. As the runtime environment supporting the execution of BPEL processes, BPEL engine should be responsive, robust and always available under massive concurrency.

Most current BPEL engine products (e.g. IBM WPS [7], Oracle BPEL Process Manager [8]) and open source projects (e.g. Apache ODE [9], ActiveBPEL [10]) adopt thread-per-request or thread pool approach to process the requests. Although threads allow a simpler and more natural programming style, thread-per-request can lead to serious performance degradation when the number of threads is large, while thread pool can introduce unfairness when all threads are busy or blocked. Furthermore, three types of concurrency in BPEL engine have magnified this performance challenge. There are concurrent requests from clients, concurrent process instances in engine and concurrent branches in a process instance. This challenge can be partly resolved by using cluster technology. Given an engine that can handle a certain number of connects, it is possible to sustain a many-fold increase in load when replicate the engine to form a cluster. However, the performance of each engine in a cluster is also critical, because it is not economical to use many low-performance engines to handle the maximum potential demand, according to the peak load may be far greater than the average. Therefore, our goal is to design a general framework for high-concurrency BPEL engine that handle load gracefully.

We follow the ideas of SEDA [11], which is a staged event-driven architecture for highly concurrent Internet services, and propose a new architecture for high-concurrency BPEL engine, called SEDA4BPEL. Our contributions include:

- We encapsulate the processing of BPEL related web services protocols (such as HTTP, SOAP [12] and WS-Security [13]) into four primary event-driven stages (Section III (A)). It provides independence, isolation and modularity for the implementations of web services
We introduce two controllers to manage excessive concurrent process instances. (Section III (B)). One of them can help to restrict the number of process instances when the load is heavy. Another can help to maintain a minimum number of necessary process instances in memory and maximize the use of system resources. We implement a BEPL engine based on SEDA4BPEL, called XServices BPEL Engine.

The rest of this paper is structured as follows: Section II provides the background and related work for the following discussion. In Section III, we discuss the SEDA4BPEL in detail. Section IV gives the implementation of XServices BPEL Engine. The experiment and the result evaluations are presented in Section V. Finally, Section VI contains our conclusions and future work.

II. BACKGROUND AND RELATED WORK

In 1978, the discussion of Lauer and Needham [14] argued that the process-based and message-passing models are dual to each other, both in terms of program structure and performance characteristics. However, it cannot stop the long-term debate between threads and events about which one is better for high-concurrency architecture [15] [16]. There are also some systems have been developed and evaluated to support the standpoints of each side, such as Capriccio [17] (thread-based) and Flash [18] (event-based).

SEDA is a staged event-driven architecture to enable high concurrency, load conditioning and ease of engineering for Internet services. It combines the thread-based model for ease of programming and event-based model for extensive concurrency. In SEDA, applications consist of a network of event-driven stages connected by explicit queues. A stage is composed of an event queue, a thread pool, an event handler and a controller, as depicted in Figure 1. This architecture allows services to be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. SEDA also makes use of a set of dynamic resource controllers to keep stages within their operating regime despite large fluctuations in load. [11] gives a more detailed introduction of SEDA.

Although SEDA is an architecture designed for Web server, we can follow its idea in designing BPEL engine based on the fact that BPEL engine is also a kind of special Web server providing BPEL services. We choose SEDA as the infrastructure of BPEL engine for three reasons. First, introducing a queue between two stages decouples their execution and provides isolation and modularity. Second, decomposing the code of engine into stages and explicit event delivery mechanisms facilitates debugging and performance analyzing of engine. Third, SEDA is platform independent and don’t need to modify or reply on any special APIs or libraries of operating system, differing from the Capriccio’s dependence on Linux, for example.

However, BPEL engine is more complex than Web server. Web server is designed to handle short-lived and stateless HTTP requests. But there are many long-running process instances in BPEL engine. Moreover, the requests of BPEL engine always contain sophisticated protocols content for web services. So our work of developing a high-concurrency BPEL engine is harder than just applying the SEDA. The researches on concurrency aspect of BPEL engine are very few. Some projects focus on the scalable service orchestration, such as OnceBPEL [19] and BUST [20]. Comparing with them, SEDA4BPEL provides the entire processing chain of BPEL engine from request to response and others only concentrate on the execution of process instances.

III. DESIGN OF THE SEDA4BPEL

SEDA4BPEL is a staged event-driven architecture for high-concurrency BPEL engine. In this section, we discuss the overall design of our SEDA4BPEL approach.

A. Stages for Web Services Protocols

BPEL is not a stand-alone protocol. The external behavior of BPEL process is defined as a composited web service. A series of web services protocols are used by BPEL process to interact with its partner services and processes. So a BPEL engine needs to consider not only the executing of BPEL but also the processing of related web services protocols. It raises a number of additional challenges for the developers to organize the implementation of different web services protocols in BPEL engine.

In SEDA4BPEL, the entire processing chain of BPEL engine is composed of four primary event-driven stages, as shown in Figure 2. Transport stage implements the protocols which responsible for transporting messages between web services, such as HTTP, SMTP and FTP. Message stage implements the protocols which responsible for encoding and reliably forwarding messages to the destinations, such as SOAP, WS-Addressing and WS-Reliability. Service stage implements the protocols which responsible for describing service, discovering service, as well as guaranteeing security and consistency, such as WSDL, UDDI, WS-Security and WS-Transactions. Orchestration stage implements the protocols which responsible for orchestrating services and
human tasks, such as WS-BPEL, BPEL4People and WS-HumanTask. Table I shows some popular web services protocols which could be encapsulated into those stages.

Excepting the orchestration stage, each of other three stages can be divided into two sub-stages: request stage and response stage. The request stage handles request message event (reqME), while the response stage constructs response message event (resME). In normal circumstances, the event handler of request stage will get a reqME from incoming event queue, process it and dispatch a reqME by enqueuing it on the event queue of next request stage or orchestration stage. If it causes some errors or exceptions when handles reqME, the event handler will create a resME containing error or exception information, and add it to the event queue of corresponding response stage. In each stage, including all the request and response stages, the concrete implementation of a special protocol could be further divided into child stages according to the need.

Encapsulating the implementation of BPEL related web services protocols into stages can bring some evident advantages:

**Independence:** The implementation of a protocol need not to rely on other protocols. It communicates with other protocols only by sending and receiving events.

**Isolation:** Other stage communicates with the implementation through its event queue, rather than by calling it directly. So the errors of this implementation can be limited to its own stage.

**Modularity:** An implementation can be easily replaced by other implementations, which may be a third-party or open-source code.

### Table I

<table>
<thead>
<tr>
<th>Stages</th>
<th>Web Services Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orchestration</td>
<td>WS-BPEL, BPEL4People, WS-HumanTask</td>
</tr>
<tr>
<td>Message</td>
<td>SOAP (SOAP with Attachments, MTOM, RRSHB), WS-Reliability, WS-ReliableMessaging, WS-Addressing, etc.</td>
</tr>
<tr>
<td>Transport</td>
<td>HTTP, HTTPS, SMTP, FTP, JMS, IIOP, etc.</td>
</tr>
</tbody>
</table>

### B. Management of Concurrent Process Instances

Each process instance represents one individual enactment of the BPEL process, using its own process instance data. The data includes variables, partner links, execution states and so on. The process instance can be long-running when the involved services might not be able to react instantly to the requests. This happens particularly in scenarios where a process instance asynchronously invokes a partner and then waits for its response. If multiple process instances are running simultaneously over a long period of time, the server resources are consumed quickly, that causes serious performance degradation.

Figure 3 shows the dispatcher sub-stage and process sub-stages in orchestration stage of SEDA4BPEL. There are one dispatcher sub-stage and some process sub-stages. Each process sub-stage corresponds to a BPEL process. When a reqME was received, dispatcher sub-stage dispatches it to a particular process sub-stage according to the process name. If the reqME matches a <receive> or a <pick> activity annotated with a createInstance="yes" attribute in BPEL process, a new instance of the process will be created by process sub-stage. Otherwise, the reqME will be added to the event queue of an existing process instance according to the value of correlation set.

We have implemented two controllers to manage excessive concurrent process instances. The first is the reqME controller, which restricts the number of process instances. When the queue length or the number of process instance exceeds some threshold, the controller will remove all the reqMEs, which can lead to create new process instance, from the event queue and cache them. So no new process instances will be created until the load dropped or some existing process instances finished. Those cached reqMEs will be re-added to event queue if the situation changed for the better, or the error resMEs will be constructed if timeout occurred. The reqME controller of dispatcher sub-stage can restrict the total number of process instances in BPEL engine, while the reqME controller of process sub-stage can restrict the number of process instances of a particular BPEL process.

The second is the dehydration/rehydration controller, which is only used in process sub-stage. The controller observes the event queue of all process instances. If there is no reqME dispatched to a process instance for a period of time, the controller will persist all of the instance-specific data to...
the persistence database and removing the process instance from memory. This called dehydration. When the process sub-stage receives a reqME for the dehydrated process instance, the controller restores the process instance from the database to memory and then continues with the execution. This called rehydration. The dehydation/rehydration controller can help to maintain a minimum number of necessary process instances in memory and maximize the use of system resources.

IV. Implementation

We have implemented a BEPL engine based on SEDA4BPEL, called XServices BPEL Engine. Our engine is implemented entirely in Java, and consists of 52861 lines of code. Figure 4 shows the components of our engine. The design of some core components has high relevance to the structure of SEDA4BPEL.

Client Channel and Service Channel: channel is the communication infrastructure between engine and outside environment. The engine receives a request from client, and then invokes the services according to the BPEL definition, finally replies the response to client. channel is built on three primary stages of SEDA4BPEL. It implements HTTP 1.1 (HTTPS) and SMTP in transport stage, SOAP 1.1 and WS-Addressing 1.0 in message stage, WSDL 1.1 and WS-Security in service stage.

Message Pool: message pool is the container of reqMEs. In message pool, all the reqMEs are classified and stored by the id of process instances that they will be sent to. That is to say, the event queues of all process sub-stages in SEDA4BPEL, are controlled and managed by message pool.

Process Instance Manager: process instance manager manages the process instances in engine. When the process instance is created, the variables are initialized in data container and activities are pushed into activity stack. The top of activity stack keeps the activity which will be executed next. This activity will be pulled out after its finishing, process instance manager implements the reqME controller and dehydration/rehydration controller in SEDA4BPEL.

Persistence Manager: persistence manager is responsible for the dehydration and rehydration. When the process instance manager has decided which process instance need to be dehydrated or rehydrated, the persistence manager will store it to database or restore it to memory.

V. Performance Evaluation

In this section, we present experimental results that compare the performance of XServices BPEL Engine and Apache ODE. Apache ODE is a popular open source BPEL engine, which adopts thread-per-instance to manage the process instances.

A. BPEL Example

We used the “Loan Approval” process in [3] as the BPEL example for our performance evaluation. In loan process, customer client send loan requests, including personal information and amount being requested. According to the amount requested and the risk associated with the customer, the process replies either a "loan approved" message or a "loan rejected" message. The low-risk customers with the amounts of less than $10,000 are approved automatically. For higher amounts, or medium and high-risk customers, the credit request needs further processing. Figure 5(a) shows the definition structure of the loan process.

For each request, the loan process uses the functionality provided by two other services. A risk assessment service is used to obtain a quick evaluation of the risk associated with the customer. A loan approval service is used to obtain assessments for medium and high-risk customers with high amount. To avoid the influence of external services, we replace one <invoke> activity (risk assessment service) with an <assign> activity, meanwhile, replace another <invoke> activity (loan approval service) with a <wait> activity. So the modified process can be executed in BPEL engine without providing runtime for external services. When the amount is more than $10,000, the process instance will wait ten minutes to simulate the time for approval. Figure 5(b) shows the definition structure of the modified loan process.

B. Benchmark Configuration

The experiments were performed with the XServices BPEL Engine and Apache ODE 1.33 (deployed in Apache Tomcat 6.0.20 with default setting) on Solaris 9 operation system. All tests use the same server hardware based on a 3.00GHz Intel Core2 Duo CPU with 4G memory and 10/100Mbit/s Ethernet interface. Sun JDK 1.6 was used as the Java platform. Four machines of 3.40GHz Intel Pentium D CPU with 1G memory were used for load generation, with each client machine using a number of threads to simulate...
actual clients. A switched fast ethernet connects the server machine to all the client machines.

The client continually requests the loan process with a random amount from $5,000 to $15,000. The proportion of low amounts (amount < $10,000) is 80 percent, and the proportion of high amounts (amount $\geq$ $10,000$) is 20 percent. If result is replied by engine, the client will sleep for a fixed time of 2 seconds before sending the next request. Otherwise, the client will wait for the timeout event and then send a new request again.

We examine the engine's ability to handle concurrent requests and concurrent process instances separately in two experiments. The first experiment evaluates engine performance under a range of client numbers. A set of simultaneous clients repeatedly request the loan process for ten minutes, where the client number is increased by 25 in each test. The second experiment evaluates engine performance under a range of process instance numbers. A set of simultaneous clients repeatedly request the loan process until the number of process instances reached 8,000.

As a concurrent performance property of BPEL engine, the average response time for clients is more meaningful and important than the throughput. In all our experiments, the average response time is calculated from the requests only with low amount.

C. Results Analysis

Figure 6 shows the average response time achieved with XServices BPEL Engine and Apache ODE in first experiment. We can see that the average response time increased along with the increase of simultaneous clients. When the number of simultaneous clients was fewer than 175, the performance of Apache ODE was better. We think the reason is that the staged design of XServices BPEL Engine lengthens the processing chain and increases the latency. When the number of simultaneous clients was over 200, the average response time of Apache ODE changed dramatically, but that of XServices BPEL Engine increased steadily. When 500 simultaneous clients were started, the average response time for Apache ODE was over 100 seconds, and over 70 seconds for XServices BPEL Engine.

Figure 7 shows the performance influenced by the number of process instances for each engine in the second experiment. The average response time of Apache ODE was shorter than XServices BPEL Engine for the same reason mentioned above when the number of process instances was fewer than 4,500. But there was a significant performance drop for Apache ODE when the number of process instances exceeded 5,000. Furthermore, as the number of process instances reached 6,000, the runtime of Apache ODE crashed. Although the performance of XServices BPEL Engine was also declined along with the increase of process instances, the system was more stable and robust. The average response time of XServices BPEL Engine was not over 50 seconds.

To achieve better performance in the second experiment, the reqME controller and dehydration/rehydration controller of XServices BPEL Engine played an important role. Table II shows the number of cached resMEs, timeout resMEs and dehydrated process instances recorded in the experiment. The threshold of reqME controller was set to 4,000. When the number of process instances exceeded 4,000, the reqME controller started to work. It cached the resMEs to wait for the finishing of some process instances or timeout, and prevented the creation of new process instance. Although
it was not fair for some clients who wanted to create new instance, it kept enough resource for running process instance. Dehydration/rehydration controller started to work when the process instance had entered the wait state for one minute. We can notice that many process instances were dehydrated from memory to database when the number of process instances was very large. It helped to clean the memory and accept more new process instance.

VI. CONCLUSION

This paper presents a new architecture for single high-concurrency BPEL engine, called SEDA4BPEL, and describes an implementation of this architecture, the XServices BPEL Engine. In SEDA4BPEL, we encapsulate the implementation of BPEL related web services protocols into four primary event-driven stages and introduce two controllers to manage concurrent process instances. We present the experiments to evaluate the performance of XServices BPEL Engine. Results show that, compared with Apache ODE, XServices BPEL Engine exhibit high performance and robustness when handling massive concurrency.

The SEDA4BPEL behaves well when dealing with concurrent process instances, but it does not consider another important concurrent factor in BPEL engine that many concurrent branches could be generated in <$flow> activities of a process instance. So we plan to address this problem in our future work, moreover, we will cooperate with CVIC SE for applying the XServices BPEL Engine to the finance domain of China.

ACKNOWLEDGMENT

This work was supported by a grant from the National High Technology Research and Development Program of China (863) No. 2007AA010301 and No. 2009AA040330, the Major State Basic Research Development Program of China (973) No. 2005CB321901.

REFERENCES


