Towards a Scalable PaaS for Service Oriented Software

Hailong Sun, Xu Wang, Minzhi Yan, Yu Tang, Xudong Liu
School of Computer Science and Engineering
Beihang University
Beijing, China
{sunhl, wangxu, yanmz, tangy, liuxd}@act.buaa.edu.cn

***Abstract*—Software developers with service oriented technologies usually put a lot of efforts to deploy and manage supporting middleware and tools. Meanwhile PaaS in cloud computing aims at provide efficient support for software developers. In the light of this consideration, we have designed and implemented Service4All, a service cloud platform targeting at improve productivity of service oriented software developers. In this paper, we describe the design of SAE, a key component in Service4All, in terms of scalability. First, we present the key technical issues and architecture design of SAE. Second, we describe a software appliance based mechanism for elastic middleware management. Third, we describe a micro-kernel based AppEngine core for efficient coordination of various components. Finally, through a real application deployed on Service4All, we demonstrate the effectiveness of our solution.**

Keywords—Service oriented computing; Cloud computing; PaaS; Scalability; Elasticity; Software appliance

I. INTRODUCTION

Service oriented computing [1] is widely adopted to realize efficient software development by reusing loosely coupled and standard interface based services. Typical service oriented software development involve a complex suite of software and tools. Installation, deployment, configuration of the supporting software consumes a lot of effort, resulting in the decreasing of development productivity. On the other hand, under the umbrella of cloud computing [2], PaaS (Platform as a Service), is devoted to facilitating efficient software development through provisioning of on-demand hosting environment and development tools. Hence a straightforward thought is to combine the two lines of technologies to improve the productivity of building service oriented software.

To realize this goal, the key point is to migrate service oriented computing implementation to cloud computing environment. According to service oriented architecture, service discovery, service development with composition and service orchestration with middleware support are three most important issues. To provide cloud support for these three aspects are core issues to implement a PaaS for service oriented software. Our Service4All[3] is a cloud computing project focusing on PaaS layer, which aims at providing a cloud platform mainly for service-oriented software developers. In general, Service4All is mainly composed of three major building blocks: *SAE* (Service-oriented AppEngine), *ServiceFoundry* and *ServiceXchange*. SAE is responsible for providing a hosting environment for developers' applications. The latest version of SAE can support the running of atomic web services, composite web services (modeling with BPMN specification) and Java Web Applications (packaged with WAR files). *ServiceFoundry* is an online development environment providing various tools for building applications with different programming models. Currently, *SeriviceFoundry* supports the invocation of atomic web services, modeling and composing of composite web services, and deployment of WAR files. Moreover, *ServiceFoundry* is well integrated with our online service repository, ServiceXchange, so as to facilitate the resuing of existing services to build composite services. One advantage of Service4All is that both SAE and *ServiceFoundry* can be easily extended to support other programming models, which enables Service4All to meet the requirements of different developers.

In fact, we also have noticed that there is similar work in this direction. For instance, IBM launches its Smart Business Development and Test Cloud [4] to implement a flexible and cost-efficient, cloud-based development and testing environment. Hajjat et al. [5] present their work on the challenges of migrating traditional enterprise application to cloud computing environment. The work in [6] is related to our work, but it concerns about the integration of grid and cloud using service oriented computing technologies. However, we have not seen much system work on implementing a PaaS specially for service oriented computing.

In this work, we are mainly concerned with the scalability perspectives of Service4All. Scalability is a critical property for cloud services since there can be unexpected large-scale concurrent requests. For instance, in the Chinese New Year travel rush of 2012, the online ticketing system(www.12306.cn) crashed soon after it was launched in January. The crashing was caused by 1.66 million daily transactions and over 1 billion visits during one peak period. With the proliferation of crowd computing, people will not only consume services but also work online, which will incur inevitable requirements for scalability. Replication [7] is a well-known technology to be used to address scalability or availability issue in service oriented applications. However, our concern is to address this issue in the combined environment of service oriented computing and cloud computing. First, a framework of on-demand provisioning of middleware is designed to meet the

dynamic changing of service deployment and access requests. Second, a micro-kernel based service oriented AppEngine core is provided to support efficient coordination of various components. Additionally, a replication based framework [8] is proposed to ensure the scalability of application services. Due to the space limitation, we will not describe service replication scheme here.

The major contributions of this work include:

- We propose a *software appliance* centric elastic middleware management framework, with which a middleware instance can be instantiated/destroyed at a dynamically chosen node in IaaS layer.
- We provide a micro-kernel based service-oriented component management framework for AppEngine Core and we implement it on the basis of Apache ServiceMix, an opensourced enterprise service bus.
- We show the effectiveness and efficiency of SAE through a Web service load testing application deployed on Service4All.

The rest of this paper is organized as follows. Section II presents the architecture of Service Oriented AppEngine and the key technical challenges; In Section III, we describe the software appliance based elastic middleware management framework and its implementation. Section IV presents the design of AppEngine Core. In Section V, we provide the design of WS-TaaS on the basis of Service4All to show the effectiveness of our solution. Finally in Section VI, we conclude this work.

II. ARCHITECTURE DESIGN OF SAE

The main objective of SAE is to provide an on-demand and dependable runtime environment for service oriented applications. With Service4All, applications are built with the online development environment provided by ServiceFoundry through service composition. As a result, an application usually exists in the format of an ensemble of several archives, such as atomic Web service archives, composite service archives and Web application archives. Each archive contains binary program code, configuration files and other files like HTML pages and server-side Web pages. Running such an application requires corresponding middleware environment to interpret and run the abovementioned archives. For instance, an atomic Web service can be run by Apache Axis2 container, and an Web application developed with Java can be run by Apache Tomcat.

Developers need to deploy their applications onto SAE first, which should be done as simple as possible, e.g. just by a mouse click. Once this is successfully done, the deployed applications can be accessed by end users immediately. As a whole, SAE provides a dynamic hosting environment for service oriented software, which is the key to implement instant deployment and running of service oriented software. Basically, SAE is responsible for managing the runtime environment on behalf of developers so that developers can focus on application logic development.

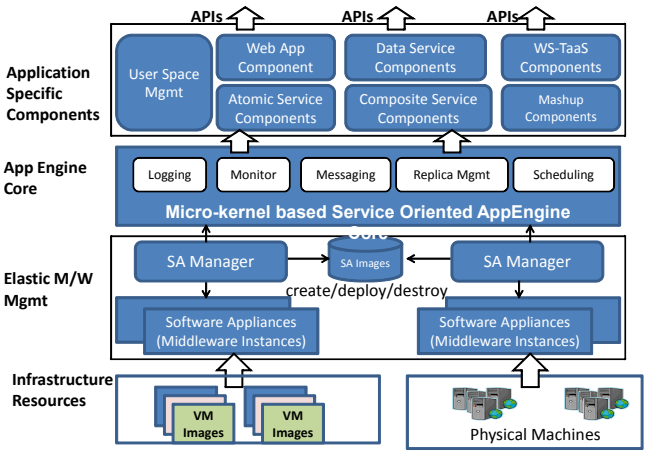

Fig1. Architecture of Service Oriented AppEngine

The main challenges that SAE faces come from three requirements. First, both the application deployment requests from developers and the end user requests arrive dynamically, an elastic middleware management is needed so as to improve the utilization of underlying resources and save costs. This means middleware should be dynamically instantiated to or destroyed from IaaS layer. Second, as the running of service oriented applications needs the coordination of multiple components, an efficient distributed mechanism of message communication among components are required. Third, when a large scale of end user requests come, the performance of corresponding application can be degraded if no effective measures are taken. Although replication is a commonly-used technique to balance the incoming load, keeping the consistency of replicas poses another challenge.

Figure 1 shows the architecture of SAE. The Infrastructural resource layer is mainly responsible for providing infrastructure resources in the form of either physical machines or virtual machines. SAE does not concern the provisioning of infrastructural resources and it simply invokes the APIs of IaaS platforms to obtain the needed underlying resources. In general, the functions of SAE can be divided into three layers as shown in Figure 1.

Elastic M/W management. The main function of this layer is to dynamically provide specific middleware instances required by upper layer. Depending on implementation solutions adopted by developers, there can involve various middleware for running an application. To simplify the management of various middleware, we propose to use *software appliance* as an abstract representation of middleware. With this abstraction, all application specific middleware is encapsulated as software appliance images with unified management interfaces such as *create*, *deploy* and *destroy*. The images are stored and managed by a SA image repository. Once receiving a middleware instance request, SA Manager is responsible for obtaining computing resources from IaaS layer and create & deploy an instance of corresponding SA image. Then the instantiated software appliance is returned to the upper layer. Additionally, when a software appliance is not needed, it should be destroyed so as to release the underlying resources.

AppEngine core. As we have pointed out before, the running of service oriented application usually needs the coordination of multiple middleware instances. Especially when new types of middleware are to be supported, communication protocols among components need to be modified correspondingly, which greatly limits the system extensibility. On the other hand, basic functions such as logging, monitoring and scheduling are accessed by all other application specific middleware instances. Therefore, efficient coordination and system extensibility are two important issues for AEC (AppEngine Core). We adopt the service oriented architecture to design AEC. Specifically, we borrow the idea of enterprise service bus to provide a uniform and loosely-coupled component coordination environment.

Application specific components. This layer lies between developers and SAE, which is composed of various types of components, which are application implementation specific. For instance, if a *.WAR file is to be deployed onto SAE, there must be a specific Web application component that is able to parse WAR files. These components must follow the specification of AEC so as to be easily integrated. With the support of these components, various APIs are abstracted and provided to developers for deploying and running applications. In SAE, the APIs are provided in the form of Web service invocation.

III. ELASTIC MIDDLEWARE MANAGEMENT

As the software that resides between applications and the underlying architecture, middleware provide support for applications to run with better performance, scalability, security, and other features. Now most of applications does not run directly on top of operating systems, instead they run with specific middleware. Applications developed with service oriented technologies usually need service container, composite service engine, enterprise service bus, Web container and so forth.

In cloud computing environment, there are three reasons why an elastic middleware management mechanism is need. First, as the number of user requests are continually changing, the load of supporting middleware also varies dynamically. When a node is overloaded, new middleware is required to share the load so as to ensure the quality of services; on the other hand, if a node is idle, the middleware at this node should be removed to make room for other overloaded applications. Second, even if load is not a problem, most of developers desire a separated runtime environment to avoid impact by other applications, which strengthen the requirements of on-demand middleware as well. Third, middleware can be subject to failure as a result of mis-behaved applications, errors of operating systems or hardware. In the event of failure, a new instance of middleware should be created.

Middleware management is usually too complex to be done manually. For instance, Apache Axis2 depends on Tomcat, and the latter depends on JVM. Deploying an Axis2 service container involves deploying three pieces of middleware and a lot of configuration work. Due to the complexity of middleware deployment, we aims at provide an automatic and elastic middleware management mechanism.

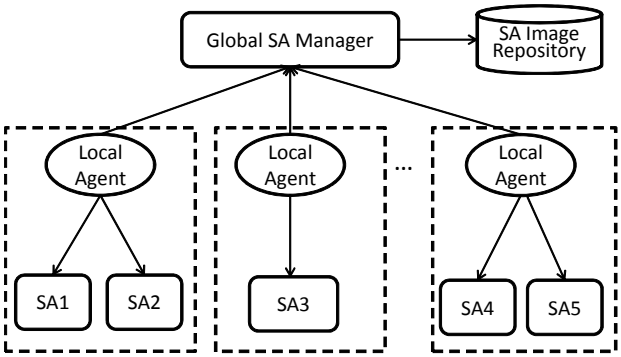

Fig 2. Design of Elastic Middleware Management Framework

The management interfaces of different middleware instances varies greatly. To reduce the complexity of managing heterogeneous middleware, we propose to define an abstraction of specific middleware, i.e. *software appliance* (*SA* for short), which is an encapsulation of certain middleware with unified management interfaces including *create*, *deploy*, *stop*, *restart* and *destroy*.

As shown in Figure 2, the software appliance based middleware management mechanism is designed in a master/slave model. At each cloud node, either a VM or a physical machine, a Local Agent is deployed to perform the management of all the SAs at that node. Global SA Manager (GSAM) is responsible for managing all the SAs through Local Agents. Each Local Agent monitors the node status and periodically report to GSAM. Once a request for a SA is received, GSAM will decide to contact which Local Agent on the basis of node status. Additionally, a repository is designed to store SA images and a Local Agent learns from GSAM about how to obtain SA images from the repository.

Once a SA is deployed, the corresponding Local Agent starts to monitor the running state of SAs through periodically heartbeat polling and report it to GSAM. When certain failures occurs, GSAM will trigger failure recovery process. Since all the information is maintained at GSAM, the recovery of a SA is the same as deploying a new one.

Figure 3 shows the implementation of GSAM and Local Agent. GSAM contains three main modules: (1) a request cache and scheduling module to schedule the request to the proper agent; (2) a global SA state storage module to fetch and update the running state of each SA from the agents; (3) a fault detection and recovery module to deal with the fault recovery tasks. And Local Agent consists of four main modules: (1) a local service repository to store the service resources to facilitate the fault recovery; (2) a request analysis and execution module to analyze and execute the request dispatched from the component; (3) a state monitor module to keep rolling and reporting the running state of local SAs; (4) a management service to provide the dynamic management interfaces of SA.

IV. SERVICE ORIENTED APPENGINE CORE

Due to the complex application logic, running an application will need the coordination of multiple components, which interact with each other more frequently. Thus it is of

great importance to design an efficient coordination infrastructure to meet the scalability requirement, which is the main objective of AEC.

First, AEC extends the microkernel architecture and design a new distributed microkernel service bus as the basic service integration environment. The service bus uses a three-tier network topology as shown in Figure 4. Each tier consists of one kind of host nodes, so there are three kinds of host nodes: *Root Node*, *Name Node* and *Exec Node*. All the messages sent to a component in the platform should be first through the Root node, and then forwarded to other nodes by *Root* node, and then passed to the specific component. Each *Name* node and the *Exec* nodes following it form a more independent self-domain, which we call functional component domain. All instances of a functional component will only exist in one domain, and this domain may have different functional components' instances. An *Exec* node mainly provides the operating environment for its instances. All functional components' instances will only run on *Exec* nodes. All messages sent to the functional components will be forwarded by *Exec* nodes to the specific instance. Eventually the specific instance responds to the request message. Service bus provides a single system image by coordinating between all these three kinds of nodes. For platform users and developers, service bus only has a minimum service set such as messaging service and component management service et al. The platform adds new services through deploying and loading new components.

Second, AEC uses the service oriented methods to model all the functions provided by components as service, then brings out a new component model, BRI Model (as shown in Figure 5) which manages the components running in the platform. In this model each component has some BusinessUnits which contain the business logic. Each BusinessUnit has one Receiver and the receivers which are the service interface of the BusinessUnit. If some BusinessUnit wants to invoke other BusinessUnits, it has to contain some Invokers that are responsible for using other BusinessUnit. These three parts are all described in standard web service way. The standard description improves the interoperation between different components and the components could easily transplant between different PaaS platforms.

Finally, we design a mechanism of dynamic component management on the basis of the above work. This mechanism aims to realize loading and unloading components dynamically in the runtime, and meanwhile not affecting other components' normal running. In order to achieve the dynamic management of the functions, the first needed work is to analyze the dependency relationships between the functional components of the platform. The dependencies are mainly the commutative callings through the interfaces between the components. Through the dependency analysis, we creates a global component dependencies list for all components running within the platform. According to the list and the possible events in the platform, we can obtain all possible states of the components and the transition model between the states. The core of dynamic management mechanism is based on such component state transition model to deploy all the components and dispatch requests efficiently.

V. CASE STUDY: WS-TAAS

In this section, we show the effectiveness and efficiency through a brief description of WS-TaaS, a real application built and running on SAE. A preliminary version of this work can be found in [9].

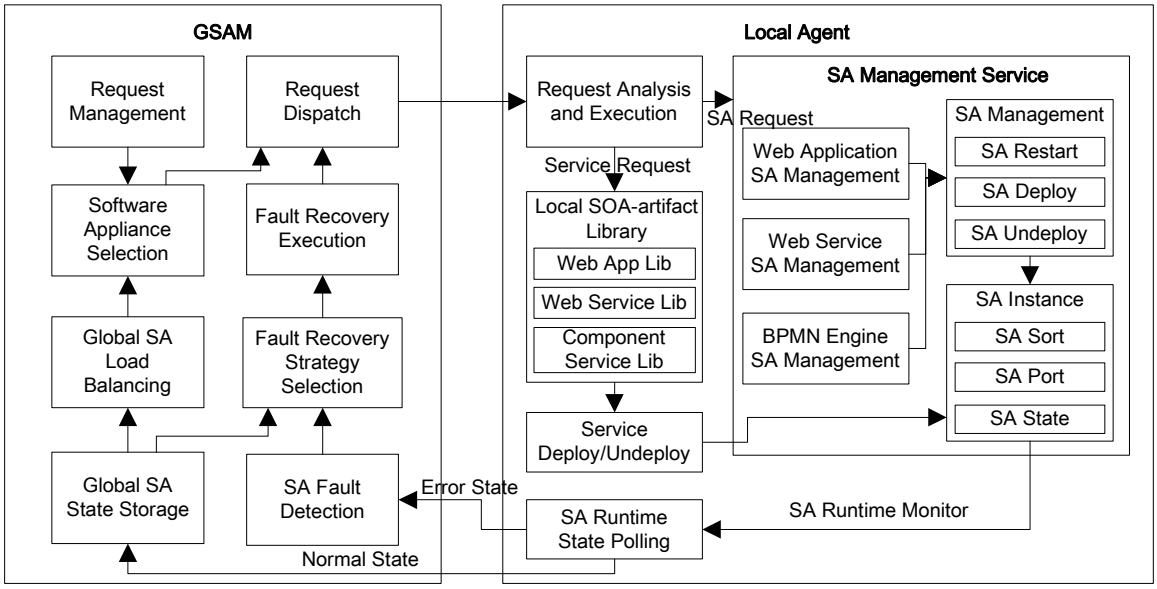

Fig 3. Implementation of SA-based Elastic Middleware Management Framework

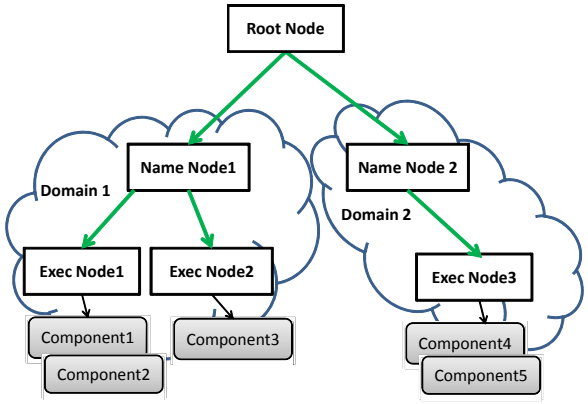

Figure 4. Distributed Micro-kernel Model of AEC

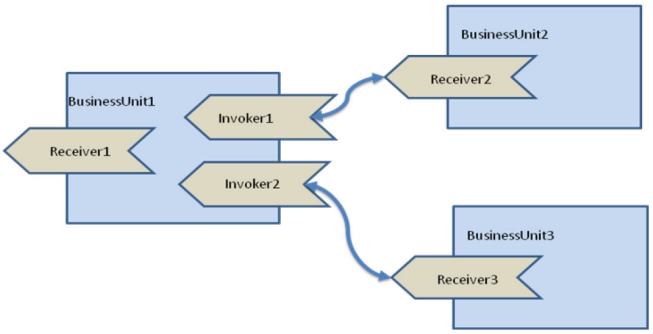

Figure 5. BRI Model

A. *Overview of WS-TaaS*

Web services are subject to a large scale of user requests, the performance of which must be fully tested before releasing because users are very sensitive to performance experience like latency. In this regards, load testing is commonly performed to estimate the performance capacity of certain software. Unlike traditional software, Web services usually runs in Internet environment and cannot be accurately tested on local computers with simulations using concurrent threads. In summary, we identify three challenges to do the load testing of a Web service: (1) simulation of real characteristics of massive user requests, including real concurrency, diversity of geographical location and system configuration; (2) flexible and elastic provisioning of testing environments, consisting of underlying resources and runtime middleware; (3) automating the load testing process.

In WS-TaaS, we address the above challenges with the help of SAE. WS-TaaS aims at providing a cloud-based Web Service load testing environment by taking advantage of the advantages of cloud testing. The features of WS-TaaS includes *transparency*, *elasticity*, *geographical distribution*, *massive concurrency and sufficient bandwidth*.

- *Transparency*. The transparency in WS-TaaS is divided into two aspects: (1) Hardware Transparency: Testers have no need to know exactly where the test nodes are deployed. (2) Middleware Transparency: When hardware environment is ready, the testing middleware will be prepared automatically.
- *Elasticity*. All the test capabilities should scale up and down automatically commensurate with the test demand. In WS-TaaS, the required resources of every test task are estimated in advance to provision more or withdraw the extra ones.
- *Geographical Distribution*. To simulate the real runtime scenario of a web service, WS-TaaS provides geographically distributed test nodes to simulate multiple users from different locations.
- *Massive Concurrency and Sufficient Bandwidth*. As in Web Service load testing process the load span can be very wide, thus WS-TaaS serves massive concurrent load testing. Meanwhile, the bandwidth is sufficient accordingly.

B. *Design and Implementation of WS-TaaS*

In a cloud-based load testing environment for Web Service, we argue that these four components are needed (as shown in Figure 6): Test Task Receiver & Monitor(TTRM), Test Task Manager(TTM), Middleware Manager and TestRunner.

- Test Task Receiver & Monitor. TTRM is in charge of supplying testers with friendly guide to input test configuration information and submitting test tasks. The testing process can also be monitored here.
- Test Task Manager. TTM manages the queue of test tasks and dispatches them to test nodes in the light of testers' intention, then gathers and trims test results.
- TestRunner. TestRunners are deployed on each test node and play the role of web service invoker, and they can also analyse the validity of web service invocation results.
- Middleware Manager. It is needed to manage all the TestRunners and provide available TestRunners for TTM with elasticity.

As shown in Figure 7, WS-TaaS is a typical three-layer structure of cloud which consists of SaaS, PaaS and IaaS layer. On the basis of Service4All, we develop three new modules and extend two existing ones in WS-TaaS. Three modules in Figure 7 are newly designed ones, including Web Service LoadTester in SaaS layer, Test Task Manager (TTM) and a new type of Software Appliance TestEngine in PaaS layer. SA Manager and Local Agent deployed in every node are also extended to support the registration and management of TestEngine.

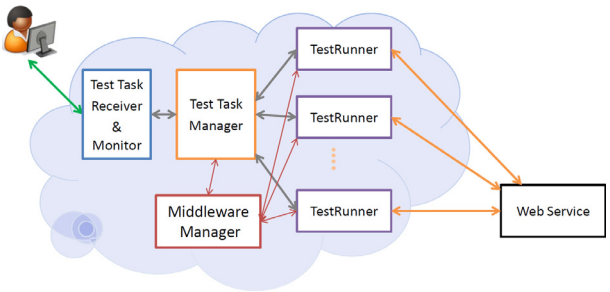

Fig 6. Conceptual Architecture of WS-TaaS

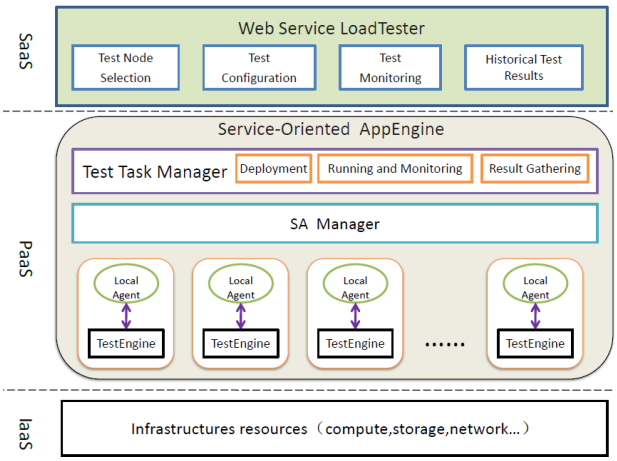

Fig 7. System Architecture of WS-TaaS

C. *Experimental results*

To evaluate the effectiveness of WS-TaaS, we choose to test the load capacity of three real Web services. To simulate the real service environment, we utilize the computing nodes from Planetlab. PlanetLab currently consists of 1,172 nodes at 552 sites around the world. We have applied for 959 slices for deploying WS-TaaS on PlanetLab and now use 50 slices as test nodes. The experiments in this section are mainly finished on these PlanetLab slices, and some test nodes in our lab, which locates in Beijing, China, are also used.

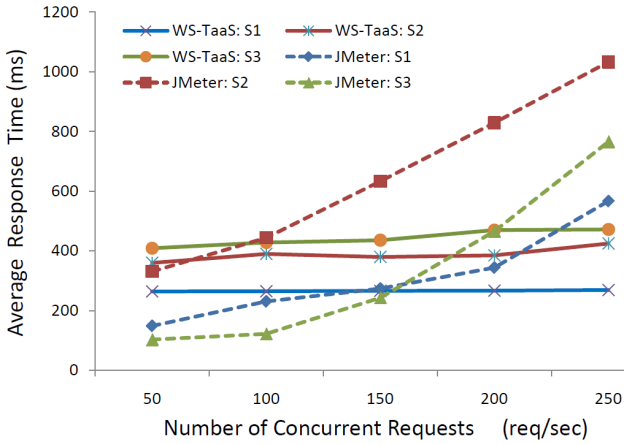

Fig. 8 ART Comparison Under Diverse Load

In this experiment, we choose three services(S1, S2, S3) as the target services to compare the performance of traditional single node web service load testing approach(JMeter) with that of WSTaaS. The node for running JMeter locates in our lab, and we choose 10 PlanetLab nodes to serve as WS-TaaS test nodes. Figure 8 displays the results of these services which show that along with the increase of the concurrent request number, the ART of JMeter grows quickly while those of WS-TaaS grow at very slow rates for S1, S2 and S3. Under the load of 250 concurrent requests, the ART of JMeter for S2 is more than 600 ms higher than the WS-TaaS ART for the same service. We could make the conclusion that the ART difference under heavy load mainly results from the limited bandwidth and computing capability of the sole node, which leads to the queuing of multiple threads. Nevertheless, instead of queuing, test threads in WS-TaaS can be scheduled more efficiently.

VI. Conclusion and Future Work

Service oriented computing has been widely accepted in real application development. Although it greatly improves the efficiency of application development by reusing existing services, the complexity behind development and running service oriented applications has not attracted enough consideration. We design and implement a PaaS platform, Service4All, specially for reduce the burden of developers with service oriented technologies. In this work, we are mainly concerned with scalability issue which is of paramount importance for PaaS platform. We describe our elastic middleware management framework and micro-kernel based service oriented AppEngine core in detail. And we show the effectiveness of our solution with a real application, WS-TaaS.

Future work leads to two directions. First, we plan to perform large-scale performance test of Service4All, and improve our methods based on the evaluation results. Second, crowd computing has been gaining increasing attention and the large number of people will incurs huge overhead to the platform. We plan to investigate how to provide platform support for crowd computing especially in terms of scalability.

Acknowledgment

This work was supported by China 863 program (No.2012AA011203), National Natural Science Foundation of China(No.61103031, No.61370057), A Foundation for the Author of National Excellent Doctoral Dissertation of PR China and Beijing Nova Program.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," in IEEE Computer. vol. 40 (11), 2007, pp. 64-71.

[2] M. Armbrust, A. Fox, et.al. "Above the clouds: A Berkeley view of cloud computing". Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[3] Service4All: http://115.28.38.203/repository/.

[4] IBM Smart Business Development and Test Cloud. http://www-935.ibm.com/services/us/index.wss/offering/midware/a1030965.

[5] M. Hajjat, X. Sun et. al. “Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud”. SIGCOMM 2010.

[6] Ashiq Anjum, Richard Hill et. al. Glueing grids and clouds together: a service-oriented approach. Int. J. of Web and Grid Services, 2012 Vol.8, No.3, pp.248 – 265.

[7] SALAS J, PEREZ-SORROSAL F,et.al. WS-replication: a framework for highly available web services. WWW 2006.

[8] Xu Wang,Hailong Sun et.al. Rep4WS:A Paxos based Replication Framework for Building Consistent and Reliable Web Services. ICWS 2012.

[9] Minzhi Yan,Hailong Sun, Xu Wang, Xudong Liu. WS-TaaS: A Testing as a Service Platform for Web Service Load Testing. IEEE ICPADS 2012