**Service4All: a scalable PaaS platform for service-oriented software**

Hailong Sun*, Xu Wang, Minzhi Yan, Yu Tang and Xudong Liu

School of Computer Science and Engineering, Beihang University, Beijing 100191, China
Email: sunhl@act.buaa.edu.cn
Email: wangxu@act.buaa.edu.cn
Email: yanmz@act.buaa.edu.cn
Email: tangyu@act.buaa.edu.cn
Email: liuxd@act.buaa.edu.cn
*Corresponding author

**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 providing efficient support for software developers. In the light of this consideration, we have designed and implemented Service4All, a service cloud platform targeting at improving productivity of service-oriented software developers. In this paper, we describe the key technical issues, architecture design and implementation technology of SAE, a key component in Service4All, in terms of scalability. First, we propose a software appliance-based mechanism for elastic middleware management. Second, we describe a micro-kernel-based AppEngine core for efficient coordination of various components. Third, we present a scalable replication framework to achieve highly scalable application services. Finally, through a real application deployed on Service4All, we demonstrate the effectiveness of our solution.

**Keywords:** service-oriented computing; cloud computing; web service; PaaS; Service4All; scalability; elasticity; consistency; software appliance.


**Biographical notes:** Hailong Sun is an Associate Professor in the School of Computer Science and Engineering, Beihang University, Beijing, China. He received the BS degree in Computer Science from Beijing Jiaotong University in 2001. He received the PhD degree in Computer Software and Theory from Beihang University in 2008. His research interests include services computing, cloud computing and distributed systems. He is a member of the IEEE and the ACM.

Xu Wang received the BS degree from Beihang University, China, in 2008. Currently, he is working towards the PhD degree in the School of Computer Science and Engineering, Beihang University, Beijing, China. His research interests include the areas of distributed systems, service computing and cloud computing.
Minzhi Yan received the BS degree from Beihang University, China, in 2010. He is working towards the PhD degree in the School of Computer Science and Engineering, Beihang University, Beijing, China. His research interests include the areas of crowd computing, software testing and service computing.

Yu Tang received the BS degree from Beihang University, China, in 2011. He is currently working towards the PhD degree in the School of Computer Science and Engineering, Beihang University, Beijing, China. His research interests include the areas of databases, distributed systems and cloud computing.

Xudong Liu is a Professor and Doctoral Supervisor at Beihang University. He received the PhD degree in Computer Application Technology from Beihang University, Beijing, China. His research interests mainly include middleware technology and applications, service-oriented computing, trusted network computing, cloud computing and crowd computing.

This paper is a revised and expanded version of a paper entitled ‘Towards a scalable PaaS for service oriented software’ presented at Proceedings of the 1st Crowd and Cloud Computing Workshop in Conjunction with ICPADS’2013’, Seoul, Korea, 15 December 2013.

1 Introduction

Service-oriented computing (Papazoglou et al., 2007) is widely adopted to realise efficient software development by reusing loosely coupled and standard interface-based services. Typical service-oriented software development involves a complex suite of software and tools. Installation, deployment and configuration of the supporting software consume a lot of efforts, resulting in the decreasing of development productivity. On the other hand, under the umbrella of cloud computing (Armbrust et al., 2009), 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 realise this goal, the key point is to migrate service-oriented computing implementation to cloud computing environment. According to Service-Oriented Architecture (SOA) 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 (Service4All, http://www.service4all.com.cn) 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 (modelling 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, ServiceFoundry supports the invocation of atomic web services, modelling and composing of composite web services, and
Service4All: a scalable PaaS platform for service-oriented software

deployment of WAR files. Moreover, ServiceFoundry is well integrated with our online service repository, ServiceXchange, so as to facilitate the reusing 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 (http://www-935.ibm.com/services/hk/en/it-services/smart-business-development-and-test-cloud.html) to implement a flexible and cost-efficient, cloud-based development and testing environment. Hajjat et al. (2010) present their work on the challenges of migrating traditional enterprise application to cloud computing environment. However, we have not seen much system work on implementing a PaaS, especially 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 (Goel et al., 2005; Salas et al., 2006; Taniar et al., 2008) is a well-known technology to be used to address scalability or availability issue in distributed systems. 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. Third, a scalable replication-based framework is proposed to ensure the scalability of application services. The major contributions of this work are followings:

- We propose a software appliance-based 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 open source enterprise service bus;
- We present a scalable replication framework to achieve highly scalable application services by leveraging a dynamic reverse proxy extended from Nginx and a Paxos-based replication protocol;
- We show the effectiveness and efficiency of SAE through a web service load testing application deployed on Service4All.

The rest of this paper is organised as follows: Section 2 introduces the overview of Service4All, and Section 3 presents the architecture of service-oriented AppEngine and the key technical challenges. In Section 4, we describe the software appliance-based elastic middleware management framework and its implementation. Section 5 presents the design of AppEngine Core. A scalable replication framework is described in Section 6. In Section 7, we provide the design of WS-TaaS on the basis of Service4All to show the effectiveness of our solution. Section 8 presents the related work. Finally in Section 9, we conclude this work.
2 Overview of service4all

The service-oriented architecture (Papazoglou et al., 2007) shows that service discovery, service development with composition and service orchestration with middleware support are three most important issues in service-oriented software development. In order to provide cloud support for these three aspects, we design and implement Service4All to facilitate the development of service-oriented software, which support service search based on collection and analysis of service resources, online development through browser-based service composition tools, and scalable composite service runtime environment with dynamic middleware management. The corresponding three pillars of Service4All are: ServiceXchange, ServiceFoundry and SAE, which are shown in Figure 1.

Figure 1  System architecture of Service4All (see online version for colours)

In Service4All, ServiceXchange, ServiceFoundry and SAE are seamlessly integrated according to the development process of service-oriented software. Services discovered in ServiceXchange can be subscribed directly to user spaces in ServiceFoundry, which are viewed as atomic services and can be assembled as composite services by online development tools of ServiceFoundry. When composite services are developed, they can be transparently deployed onto underlying on-demand provisioned middleware in SAE and then deliver services through SAE.

2.1 ServiceXchange

ServiceXchange is a platform to collect services on the web, to analyse the relationships of them, and to continuously monitor their responses for fetching Quality of Service (QoS) information. Developers can search services by keywords, names, tags and even potential service relations with the QoS requirements.
Service4All: a scalable PaaS platform for service-oriented software

Now there are about 30,000 web services collected in ServiceXchange. Developers can find their required services and then subscribe and reuse them to build complex service-oriented applications. Although ServiceXchange has been integrated with Service4All, it can still work as a stand-alone platform that supports service discovery for external users.

2.2 ServiceFoundry

ServiceFoundry provides an online development environment for service-oriented software with a set of browser-based developing tools. Currently, it supports two programming models for service composition: business process-based service composition in compliance with Business Process Modelling Notion (BPMN) specification and data centric service mashup. Therefore, ServiceFoundry mainly includes Business Process Integrated Development Environment (BPIDELite) and a Mashup Editor called Mashroom for supporting these two programming models.

In addition, ServiceFoundry encapsulates the deployment interfaces of atomic services, composite services (described by BPMN) and java web applications exposed by SAE into corresponding user-friendly deployment tools, which can support completely transparent deployment of these kinds of applications.

2.3 Service-oriented AppEngine

Service-oriented AppEngine is an online runtime environment for atomic services, composite services and java web applications. When an application is deployed onto SAE, the deployed file will be automatically transferred to on-demand provisioned middleware containers. The selection, the remote copy and the instantiation of middleware, and the transferring of the deployed file are all automatic and transparent. The middleware currently supported by SAE includes atomic service container (Apache Axis2), composite service engine (BPMN Engine), and java application server (Apache Tomcat).

Scalability has become a critical property for cloud services and is much challenging, thus we are concerned with the scalability of Service4All. Although ServiceXchange and ServiceFoundry may encounter massive requests from developers, they have relatively lighter load comparing with SAE since all atomic services, composite services and java web applications are deployed in SAE and these services or applications may be accessed by large-scale end users. Therefore, we are focusing on the design and implementation of SAE in the following sections.

3 Architecture design of SAE

The main objective of SAE is to provide a scalable 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 such as HTML pages and server-side web pages. Running such an
application requires corresponding middleware environment to interpret and run the above-mentioned archives. For instance, an atomic web service can be run by Apache Axis2 container, and a 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.

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, thus an elastic middleware management is needed so as to improve the utilisation 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 is 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 2 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 2.
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 an 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.

4 Elastic middleware management

As the software that resides between applications and the underlying architecture, middleware provides support for applications to run with better performance, scalability, security and other features. Now most of applications do 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 QoSs; 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 misbehaved 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 works. Owing to the complexity of middleware deployment, we aim at providing an automatic and elastic middleware management mechanism.

The management interfaces of different middleware instances vary 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 3, 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 an 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.

Figure 3  Design of elastic middleware management framework

Once an 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 occur, GSAM will trigger failure recovery process. Since all the information is maintained at GSAM, the recovery of an SA is the same as deploying a new one.

Figure 4 shows the implementation of GSAM and local agent. GSAM contains three main modules: (a) a request cache and scheduling module to schedule the request to the proper agent; (b) a global SA state storage module to fetch and update the running state of each SA from the agents; (c) a fault detection and recovery module to deal with the fault recovery tasks. And local agent consists of four main modules: (a) a local service repository to store the service resources to facilitate the fault recovery; (b) a request analysis and execution module to analyse and execute the request dispatched from the
Service4All: a scalable PaaS platform for service-oriented software

component; (c) a state monitor module to keep rolling and reporting the running state of local SAs; (d) a management service to provide the dynamic management interfaces of SA.

Figure 4  Implementation of SA-based elastic middleware management framework

5  Service-oriented AppEngine core

Owing 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 micro-kernel architecture and design a new distributed micro-kernel service bus as the basic service integration environment. The service bus uses a three-tier network topology as shown in Figure 5. 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. 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 6, 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 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 BusinessUnits. 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.

Figure 5  Distributed micro-kernel model of AEC (see online version for colours)

Figure 6  BRI model (see online version for colours)
Finally, we design a mechanism of dynamic component management on the basis of the above work. This mechanism aims to realise 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 analyse 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 create a global component dependency 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.

6 Scalable replication framework

On the basis of elastic middleware management and efficient component management, a scalable replication framework (Wang et al., 2012; Yuan et al., 2013) is proposed in Service4All to ensure the scalability of application services. Figure 7 shows that our scalable replication framework mainly includes the dynamic reverse proxy and the Paxos-based replication protocol. The dynamic reverse proxy can be used to scale service replicas; and the Paxos-based replication protocol can guarantee the consistency of service replicas.

Figure 7  Scalable replication framework (see online version for colours)
Dynamic reverse proxy: In Service4All, we extend reverse proxy tools such as (http://nginx.org) to support scalable replicas and load balance among them. When deploying/scaling an application service in Service4All, a deploy/scale request is sent to the deploy/scale module which is usually embedded in one application specific component. Then, the deploy/scale module will request the elastic middleware management to allocate SAs and instantiate the service replicas. As a result, the access addresses of these service replicas will be returned and sent to the dynamic reverse proxy. Finally, the dynamic reverse proxy reconfigures itself according to the new address list and reloads. After successful reload, the mapping relationship between the reverse proxy and back-end service replicas will be established. If users invoke a service, the reverse proxy will receive requests, and one of the back-end service replicas will be selected based on a certain load balancing strategy predefined in the reverse proxy. In this way, the dynamic reverse proxy enables service replicas to scale out and balance load efficiently.

Paxos-based replication protocol: In order to ensure the consistency of service replicas, we propose a Paxos-based replication protocol among replicas. Since our replication protocol is built on the basis of Paxos, a PaxosNode service is implemented and deployed onto every SA. These PaxosNode services constitute a replication configuration containing their addresses. For all invocation requests of service replicas, all replicas run the Paxos protocol to ensure that requests are executed equivalently in the same sequence so as to maintain consistency. In addition, PaxosNode services also detect failures of replica nodes and can recover failures once some replica nodes crash; therefore, the aggregate availability of corresponding application services can always maintain a reasonable level.

7 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 the work of Yan et al. (2012).

7.1 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 regard, load testing is commonly performed to estimate the performance capacity of certain software. Unlike traditional software, web services usually run 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: (a) simulation of real characteristics of massive user requests, including real concurrency, diversity of geographical location and system configuration; (b) flexible and elastic provisioning of testing environments, consisting of underlying resources and runtime middleware; (c) 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 include transparency, elasticity, geographical distribution, massive concurrency and sufficient bandwidth.
**Service4All: a scalable PaaS platform for service-oriented software**

- **Transparency.** The transparency in WS-TaaS is divided into two aspects: (a) hardware transparency: testers have not needed to know exactly where the test nodes are deployed and (b) 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.

### 7.2 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 8): 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.

**Figure 8** Conceptual architecture of WS-TaaS (see online version for colours)
As shown in Figure 9, 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, 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.

**Figure 9** System architecture of WS-TaaS (see online version for colours)

### 7.3 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 utilise the computing nodes from Planetlab. PlanetLab currently consists of 1172 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.

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 ten PlanetLab nodes to serve as WS-TaaS test nodes. Figure 10 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.

Figure 10  ART comparison under diverse load (see online version for colours)

8 Related work

8.1 SOA and cloud computing

Several works have been done to analyse how to combine Service-Oriented Architecture (SOA) and cloud computing, and how one of them influences each other. Hirzalla (2010) discusses how to realise business agility requirements through the potential synergies between SOA and cloud computing. Feuerlicht and Govardhan (2009) show that the scope of SOA needs to be extended to encompass different types of service models driven by cloud computing. Tsai et al. (2010) propose a Service-Oriented Cloud Computing Architecture (SOCCA), so that clouds can interoperate with each other, and also present high level designs to better support multi-tenancy feature of cloud computing. Azeez et al. (2010) present an architecture for achieving multi-tenancy at the SOA level, which enables users to run their services and other SOA artefacts in a multi-tenant SOA framework as well as provide an environment to build multi-tenant applications. Some other works (such as Flahive et al., 2013a; Flahive et al., 2013b) focus on how to deliver specific services with clouds.

All above works propose methods to incorporate SOA to cloud computing, discuss the impact of cloud on SOA and the synergies between SOA and cloud computing, but less leverages cloud computing technology to enhance SOA and especially the service-oriented software development.
8.2 Popular PaaS platforms


GAE enables the development and runtime of web applications based on Java and Python. Developers need to install a local GAE-specific SDK or Eclipse-Plugin to develop and test their applications; Amazon AWS Elastic Beanstalk provides the middleware stack support for PHP, Python, Ruby, .NET and Java, which requires developers to upload applications by Git, Eclipse and Visual Studio; Windows Azure facilitates the development of .NET, PHP, Java and so on, which can be deployed through Eclipse and Visual Studio tools. In addition, it provides AppFabric to coordinate multiple applications; IBM Smart Business Development and Test Cloud can manage flexible development and testing private cloud environments to help save operating costs by on-demand provisioning of physical and virtualised test resources such as operating systems, middleware, storage, network; and Cloud Foundry as an open-source PaaS provides a runtime environment to deploy, update and scale web applications, which use an extensibility technique called buildpacks to support almost all popular programming languages and frameworks.

Although these representative PaaSs are useful to host web applications coded by common programming languages and frameworks, few of them provide online development tools and the service-oriented software development model has not been supported.

9 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, especially 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, micro-kernel-based service-oriented AppEngine core and scalable replication framework 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.
Acknowledgements

This work was supported by National Natural Science Foundation of China (No. 61370057), China 863 program (No. 2012AA011203), a foundation for the author of National Excellent Doctoral Dissertation of PR China (No. 201159), Beijing Nova Program (2011022), and Program for New Century Excellent Talents in University.

References


