An extreme automation framework for scaling cloud applications

J. Yang
T. Yu
L. R. Jian
J. Qiu
Y. Li

The elastic cloud-computing infrastructure, as well as its pay-as-you-go price model, attracts increasingly more enterprises to deploy their applications in the cloud. However, it is nontrivial to scale applications automatically due to the dynamic nature of the cloud-computing infrastructure and the dependencies among application components. The challenges include declaration of extensible scaling rules to satisfy application-specific requirements, the coordination of scaling actions that may interfere with each other, and the resolution of dynamic information that can only be determined during runtime. To address these challenges, we designed and implemented an extreme automation framework, which enables the autoscaling capability of applications by automatically carrying out user-specified scaling policies during runtime. The contribution of the extreme automation framework is twofold. First, it alleviates application administrators' burden of making the right scaling decisions. Second, it helps application administrators to coordinate scaling actions to avoid potential resource contention. The proposed framework has been fully implemented and verified with different types of cloud applications, including web applications hosted by Tomcat™ clusters and WebSphere® application server clusters, Web 2.0 applications hosted by sMash clusters, and map-reduce applications deployed in Hadoop™ clusters.

Introduction

Most web-based enterprise applications are potentially targeted to users around the world, and it is very difficult to predict the access pattern of enterprise applications before they are ready to serve users. For example, the famous Facebook** application, "Friends for Sale" (which is a massively multiplayer online game that allows players to buy and sell virtual pets representing other players), has grown at a monthly rate of 300%, which surprised even its founder [1]. Some social networking sites have grown even more rapidly when they expanded registration to include worldwide users [2]. To cope with peak workload on demand and anticipate potentially high growth rates, enterprises often overprovision their servers by as much as 500% [3], which may lead to a considerable waste of resources for normal workloads and increased management costs spent on the overprovisioned infrastructure.

In recent years, cloud computing has emerged as a new paradigm for hosting information technology (IT) services over the Internet [4]. From the cloud users' perspective, the cloud provides a seemingly unlimited resource pool. Users can apply for infrastructure from the cloud pool at any time and use it through the network; they will be charged only for the actual resources used per unit time. The elasticity of the cloud-computing infrastructure and the pay-as-you-go price model attracts increasingly more enterprises to deploy their applications to a minimal set of cloud-computing infrastructures, which will later be scaled in or out to cope with workload fluctuation. In this paper, the terms *scale in* and *scale out* refer to the addition or removal of virtual machines (VMs) hosting running applications that are properly configured to work in clusters.

It is inefficient and error prone for enterprises to scale applications manually. For example, consider web applications deployed in a typical three-tier architecture (see **Figure 1**), in which the application administrator

Digital Object Identifier: 10.1147/JRD.2011.2170929

0018-8646/11/$5.00 © 2011 IBM

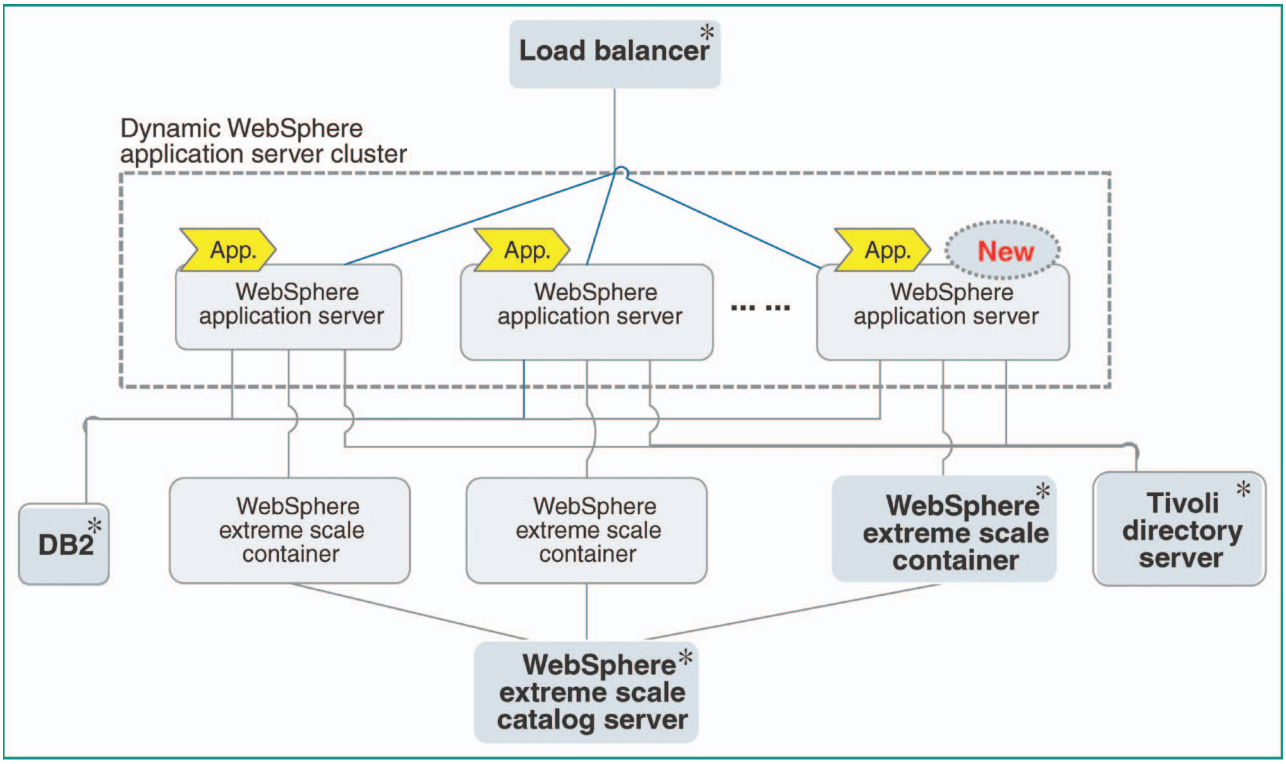

Figure 1

Typical scalable environment for web applications. The * symbol denotes components that are influenced by scaling. The label *New* refers to the fact that the component is the newly created one due to the scaling-out action. (App.: application.)

scales out the application server cluster by triggering provisioning of a new VM, manually deploying application packages to the new instance, synchronizing its state through a state server (the WebSphere* extreme scale), and registering the new instance into the load balancer. Aside from this, the administrator must still constantly monitor the runtime workload to determine the correct timing for scaling. Any human error may lead to the instability or even the crash of the whole environment. Therefore, scaling applications automatically become a natural requirement.

To automatically scale applications in a cloud-computing environment, we are confronted with several challenges, including how to support diverse application-specific scaling rules, how to coordinate correlated scaling actions with interferences, and how to resolve different types of dynamic information that is only available during runtime.

The challenges of autoscaling in a cloud-computing environment have attracted significant research efforts, from both industry and academia. Many of these efforts (e.g., [5] and [6]) focus on autoscaling at the infrastructure level, leaving application deployers and developers to ensure that their applications can work in a dynamic infrastructure. There are also solutions and research work that provide application-level scaling capability. However, these approaches have their own limitations, including inflexible scaling policies (e.g., see [7–9]), restricted types of scalable applications (e.g., [10]), and partial automation (e.g., [11]), which do not make them easily and widely usable.

The limitations of the existing work motivated us to design an extreme automation framework (EAF) for autoscaling applications in a cloud-computing environment. This EAF allows scalable clusters to declare application-specific scaling policies, which will be verified by the framework during runtime. It also provides centralized management of scaling policies to facilitate the coordination of correlated scaling actions with interferences. The resolution of runtime data is deferred to the time when the data are required by scaling actions. To avoid being a bottleneck, the EAF is self-managed and automatically scales according to the number of clusters that are registered in the automation framework.

The main contribution of the EAF is twofold. First, it alleviates application administrators' burden of making

the right scaling decisions. Second, it helps application administrators to coordinate scaling actions to avoid potential resource contention. Generally speaking, this EAF allows enterprises to migrate their applications as-is to the cloud-computing platform, with little effort required to reuse common autoscaling policies or declare application-specific autoscaling policies.

In the remainder of this paper, we first exemplify the autoscaling requirements, followed by the architecture of the extreme automation framework. Specifically, we use three paper subsections to show how autoscaling requirements are satisfied by delegated rule verification, centralized action coordination, and just-in-time data resolution. Then, we verify the feasibility and scalability of the EAF by real-world experiment and simulation. Finally, we summarize related work and conclude this paper.

Autoscaling requirements and challenges

The on-demand resource usage and pay-as-you-go price model of cloud-computing attract increasingly more users to deploy their applications in a cloud-computing environment. A typical enterprise application consists of an application server cluster, load balancer, database, directory server, and so on (see Figure 1). A natural requirement is to initially use just enough resources and apply for more only when the runtime workload increases or free resources when workload decreases. In general, the application should be automatically scaled for ever-changing workloads to ensure quality of service without a waste of resources, which is nontrivial due to the following three aspects.

First, diverse autoscaling rules must be supported. Most kinds of clusters can be scaled in a cloud-computing environment. For example, the application server may be scaled to handle more requests using request response time as a criterion; the load balancer may be scaled to accept bursts of incoming requests using incoming request rates as a criterion; the extreme-scale container may be scaled to store more in-memory data using memory utilization as a criterion. There may be more types of clusters with unpredictable application-specific criteria from the perspective of the cloud-computing platform provider. Therefore, a powerful and extensible way to specify and verify scaling rules becomes essential.

Second, scaling actions must be properly coordinated so that the whole system remains stable and consistent during scaling. In an application such as the one shown in Figure 1, scaling of the application server cluster will lead to multiple correlated actions, including registering a new application server, deploying applications into new application servers, tuning back-end databases to accommodate more concurrent database operations, and notifying a load balancer to dispatch requests to new cluster members when they are ready. At the same time, the load balancer may also need to scale to accept bursts of incoming requests. Without proper coordination, it is likely that autoscaling of different clusters leads to a chaotic entire system.

Finally, the whole application scaling process must be fully automated, including the collection of runtime workloads, the detection of threshold violation, the provisioning and configuration of new application servers, the deployment of applications in new application servers, and the coordination of correlated actions. Automation is challenging because of the dynamic nature of the cloud-computing infrastructure. The information on running clusters is ever changing: the Internet Protocol (IP) address of a new resource cannot be assigned until it is provisioned, and the cluster member that is to be removed to save cost can be deprovisioned only after all existing requests are handled. Additionally, users' anticipated scaling behavior may also change. For example, users may change the central processing unit (CPU) thresholds to scale a cluster or adjust the minimum size of existing clusters. Therefore, a mechanism is required to resolve dynamic information from the runtime environment and users.

Extreme automation framework

Overview

The EAF is an extension to our previous research work on building a scalable cloud-computing platform for scalable applications [12]. In our previous work, a simple component is responsible for parsing and verifying cluster-specific autoscaling rules. That single component introduces a single point of failure and is not scalable when there are hundreds of clusters that should be automatically scaled. This framework is targeted on providing a shared autoscaling service to enable the autoscaling capability of a large volume of clusters. The EAF itself is treated as a scalable application in a cloud and, thus, is defined as a scalable topology (see Figure 1 in [12]) in the scalable cloud platform. Specifically, the EAF is used to verify scaling rules periodically and perform cluster-specific scaling actions (as defined in the components in [12]) when scaling rules are satisfied.

The architecture of the EAF is shown in **Figure 2**. Note that only the internal architecture of an EAF instance is shown in this figure; the EAF can be made a cluster by grouping multiple EAF instances with shared repositories. After the EAF starts, every scalable cluster will register itself into the EAF after its provisioning finishes. As part of registration, scaling policies (see **Figure 3**) (including scaling rules and scaling actions) will be collected and stored in a shared policy repository. Additionally, the deployment (which is a cohesive set of clusters for a scalable application) is also saved in a shared repository. During runtime, the EAF will initiate an independent "PolicyInterpreter"

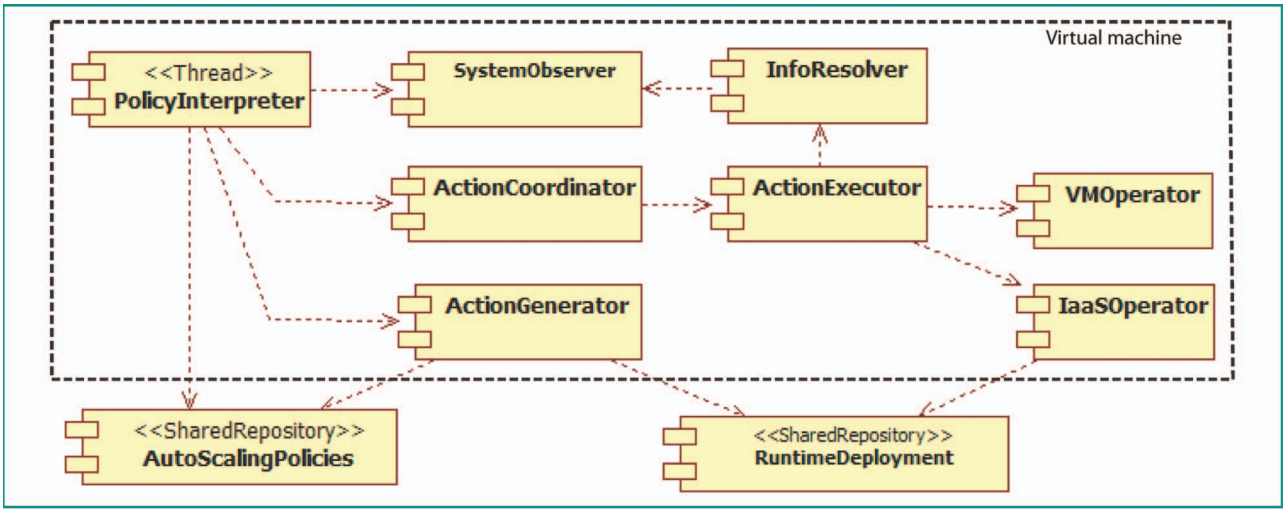

Figure 2

Architecture of the extreme automation framework.

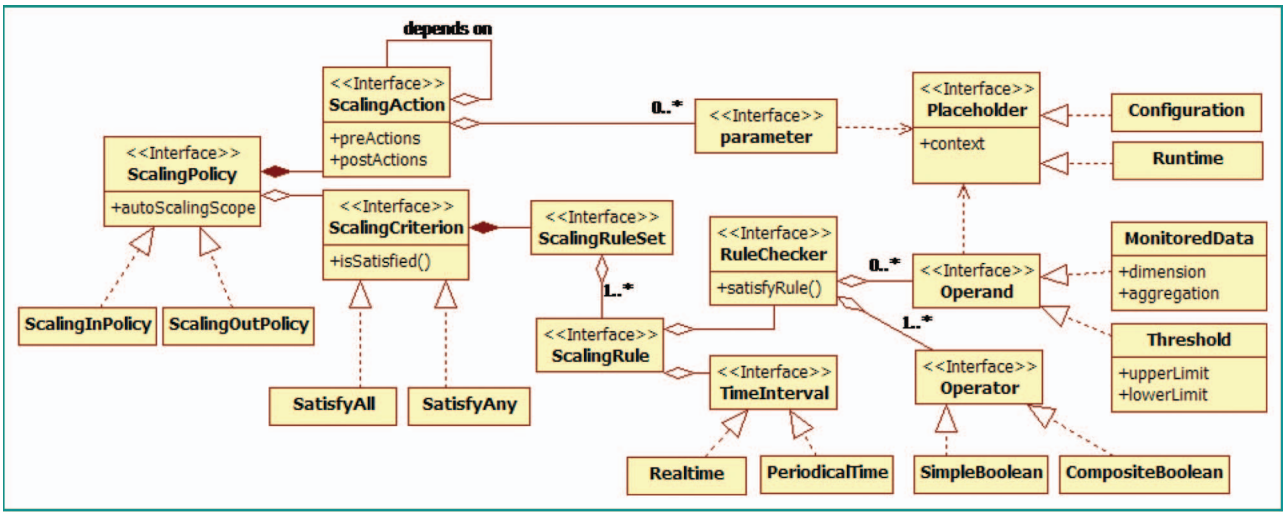

Figure 3

Unified modeling language class diagram of autoscaling policy for the extreme automation framework.

thread for each scalable cluster to verify its scaling rules. The verification is based on runtime statistics collected by the "SystemObserver." When a scaling rule is satisfied (see "ScalingCriterion" in Figure 3), the "ActionGenerator" will return a series of scaling actions (see "ScalingAction" in Figure 3) defined in the scaling policy of the cluster and in a set of correlated clusters. The scaling actions will be delivered to the "ActionCoordinator," which will coordinate all scaling actions from different clusters to avoid potential resource contention. After an action is committed, the "ActionExecutor" will turn to the "InfoResolver" for resolving dynamic data (i.e., "Placeholder" in Figure 3) involved in the action, which is finally executed through the "IaaSOperator" or the "VMOperator." (Here, IaaS stands for infrastructure as a service.) The "IaaSOperator" is responsible for such infrastructure-level operations as provisioning or deprovisioning VMs. The "VMOperator" is responsible for executing operations inside a VM

Listing 1 Sample EAF-specific scaling-out rule.

```
def scaleOutRule(verificationTimeThreshold=1, ruleCheckingDuration=3, violationPercentage=0.3):
    """the verification time is in seconds; the duration is in minutes"""
    # the rule verification time is registered in the systemObserver, which stores all verification time records
    # we can retrieve a series of verification time record (e.g. from the last 3 min. to now) during runtime
    verificationTimeRecord=systemObserver.get("EAF", "VerificationTime", ruleCheckingDuration)
    # we can do cluster-specific filtering to the record to determine those that are longer than expected
     violationRecord=filterRecord(verificationTimeRecord, verificationTimeThreshold)
     if len(violationRecord) / len(verificationTimeRecord) > 0.3:
        return True
     else:
        return False
```

(e.g., running scripts to deploy the application package to the newly created Tomcat** server after the Tomcat cluster scales out).

The EAF is targeted toward addressing the challenges of autoscaling different types of applications hosted in a cloud-computing infrastructure. These challenges include how to support application-specific scaling rules that may be widely different from one another, how to coordinate scaling actions for different clusters that may interfere with one another, and how to resolve different types of dynamic information that can only be determined during runtime. We discuss in detail how the EAF addresses these challenges in the following subsections.

Delegated rule verification

To facilitate the scaling rule declaration, the EAF provides a set of rule templates, including real-time checking and periodical checking (in minutes) of scaling rules, as well as typical logical operators to enable composite rules (e.g., the average CPU utilization of an application server cluster falls below 20% and the number of the application servers in a cluster is greater than four). In our implementation, these rule templates are packaged as Python** (a popular scripting language) functions. What follows are part of commonly used scaling rules:

```
def averageCpuUtilizationExceedsThreshold
  (cluster = CONTEXT, threshold = 0.8,
  duration = None) : ...
def averageCpuUtilizationBelowThreshold
  (cluster = CONTEXT, threshold = 0.2,
  duration = None) : ...
def maximumCpuUtilizationExceedsThreshold
  (cluster = CONTEXT, threshold = 0.8,
  duration = None) : ...
def minimumCpuUtilizationBelowThreshold
  (cluster = CONTEXT, threshold = 0.2,
  duration = None) : ...
def averageMemoryConsumptionExceedsThreshold
  (cluster = CONTEXT, threshold = 0.8,
  duration = None) : ...
def averageMemoryConsumptionBelowThreshold
  (cluster = CONTEXT, threshold = 0.2
  duration = None) :
def maximumMemoryConsumptionExceedsThreshold
  (cluster = CONTEXT, threshold = 0.8,
  duration = None) : ...
def minimumMemoryConsumptionBelowThreshold
  (cluster = CONTEXT, threshold = 0.2
  duration = None) :
```

Rule templates can be easily composed by such operators as NOT, AND, and OR, as we demonstrate in the "Evaluation" section. Additionally, developers can even declare application-specific scaling rules. For example, the EAF itself must be scalable in order to support thousands of scalable clusters. One of the key rules to scale out the EAF involves checking of whether the scaling rule verification time for a cluster exceeds a preset threshold. When this frequently happens (e.g., when 30% of the rule verification time exceeds the threshold within 3 minutes), the EAF must scale out to ensure that all scaling rules can be verified sufficiently quickly. In this case, only the EAF "knows" what kind of runtime statistics should be collected and how to make the scaling decisions. **Listing 1** shows a sample of the EAF-specific scaling-out rule.

In summary, the scaling policy supported by the EAF allows clusters to provide application-specific operators, operands, and rule verification time intervals (by subclassing or implementing proper interfaces, as defined in Figure 3). When scaling policies for each cluster are collected and stored in the policy repository, the implementation (which is manifested as Python scripts similar to the

Listing 2 Process of delegated rule verification by "PolicyInterpreter."

```
userConfiguration=getDeploymentData("userConfigurationItems") # obtain configuration from end users
scalingPolicy=getPolicyFromRepository("clusterID") # get policies from the shared repository
scalingOutRule=scalingPolicy[0].resolve(userConfiguration) # instantiate the rule with users' configuration
if pythonExecutor.verifyRule(scalingOutRule): # verify scaling-out rule first
        # continue scaling out process
else:# verify scaling-in rule
        scalingInRule=scalingPolicy[1].resolve(userConfiguration)
        if pythonExecutor.verifyRule(scalingInRule):
                #continue scaling in process
sleep (10) # wait for a period of time in seconds before next round of verification
```

above scaling-out rule scripts) for cluster-specific rule checking is also retrieved. The EAF will execute these implementations on behalf of clusters to verify their specific scaling rules. This way, clusters delegate scaling rule verification to the EAF. **Listing 2** shows how the "PolicyInterpreter" (see Figure 2) performs the delegated rule verification. Through delegated rule verification, we can achieve an easy-to-use scaling policy with a flexible extensibility.

Centralized action coordination

Different clusters independently define scaling policies, and there may exist interfewrences among the scaling actions for different clusters. First, for clusters that have dependencies between them, their scaling actions may interfere with one another. Suppose there is a system consisting of a load-balancer cluster and multiple back-end application server clusters. The load balancer depends on application server clusters to process requests; therefore, scaling out of the application server cluster results in corresponding actions in the load-balancer cluster. In this system, when both a load-balancer cluster and application server clusters are "intent" on scaling at the same time, their scaling actions must be coordinated to avoid inconsistencies. For example, scaling out of the application server cluster leads to updating the routing rules in every load balancer, and scaling out of the load-balancer cluster will replicate existing routing rules to the newly created load balancer. If these scaling-out actions are interlaced with one another, the routing rules in every load-balancer cluster may become inconsistent.

Second, for clusters that have common dependencies to shared third-party components (which may be a single server or a cluster), their scaling actions may interfere with one another. For example, when the load balancer is shared by multiple application server clusters, the scaling action of updating routing rules for different application server clusters must be coordinated to avoid potential resource contention in the load balancer. Another example occurs when a large number of clusters are requesting new instances at the same time, and the burst provisioning requests may overload the shared infrastructure without proper request control.

Therefore, an "ActionCoordinator" (see Figure 2) is introduced to ensure all scaling actions behave as expected. The coordinator considers runtime cluster status, scaling action types (i.e., VM-level actions performed by the "VMOperator" or infrastructure-level actions performed by the "IaaSOperator" in Figure 2), and the relationships among actions. The basic coordination rules are prioritized from high to low as follows.

- The scaling rules for a cluster will not be verified when the cluster is scaling in or scaling out. Scaling rules will be verified again until all scaling actions are successfully finished to avoid duplicate verification, although the cluster itself continues serving application requests during this period.
- A scaling action can only be executed after all of its prerequisite actions and actions that it depends on are finished [12].
- VM-level actions in the same VM can only be sequentially executed to avoid resource contention.
- Infrastructure-level actions will be buffered if there are too many ongoing ones to avoid overloading the cloud-computing infrastructure. If the EAF detects a time-out for an action that is buffered for a long time, the action will be treated as stalled. In that case, the scaling rules for the according cluster will be verified again to see whether the buffered action should still be executed.
- By default, all actions are concurrently executed to improve the overall performance.
- **Listing 3** shows how the "ActionCoordinator" (see Figure 2) coordinates the execution of a scaling action.

Listing 3 Coordination of executing a scaling action.

```
def: coordinateAction(currentAction):
    targetCluster=currentAction.getOperatedCluster()
    waitUntilClusterIsReady(targetCluster) # actions will be pending if the cluster is still scaling
    currentAction.setStatus("COORDINATED")
    foreach action in getPreActions(currentAction): # ensure that all prerequisite actions are finished
        if action.isNotCoordinated():
            coordinateAction(action)
        waitUntilFinished(action)
    foreach action in getActionDependencies(currentAction): # ensure that all dependency actions are finished
        if action.isNotCoordinated():
            coordinateAction(action)
        waitUntilFinished(action)

    InfoResolver.resolveAction(currentAction) # dynamic information is resolved for this action
    if ActionExecutor.requireRequestControl(currentAction):
        #if there are too many concurrent actions in the target VM or the infrastructure
        currentAction.setStatus(None)
        Pass; #It is bypassed and will be coordinated later to avoid potential conflicts with previous actions
    elif currentAction.isVMLevelOperation():
        ActionExecutor.createVMOperator().execute(currentAction); # fork a thread to perform the action
        currentAction.setStatus("RUNNING") # the action's status can be updated inside the execution thread
    else:  # similar handling for infrastructure-level actions.
        ActionExecutor.createIaaSOperator().execute(currentAction);
        currentAction.setStatus("RUNNING")
```

Just-in-time data resolution

In order to achieve extreme automation, the EAF must automatically interpret and execute scaling policies without human intervention. Unfortunately, a significant amount of information referenced by scaling policies cannot be determined until scaling policies are verified. Such information is denoted by placeholders in scaling policies (see Figure 3); they will only be resolved when they are actually used during runtime. This is what we call just-in-time data resolution.

The information denoted by placeholders originates from users' configuration (see "Configuration" in Figure 3) and information/statistics of the runtime environment (see "Runtime" in Figure 3). A typical user configuration includes the CPU thresholds to scale a cluster, the minimum size of existing clusters, and other application-specific settings (e.g., the server port, the timeout setting, etc.). Such configurations may be even adjusted by users during runtime. Typical information/statistics of the runtime environment includes the IP address of a server that is available only after the provision is finished, the workload of a server that is changing from time to time, and other cluster-related information (e.g., the current cluster size, the cluster-wide statistics, etc.).

The following code snippet exemplifies the usage of placeholders in the declaration of a scaling action. This action is used to add a new Tomcat instance into the front-end load balancer after the Tomcat cluster scales out:

```
<scaleOutAction forCluster = ''Tomcat''
purpose = ''to add new Tomcat instance into
load balancer''>
    <actionScript residesIn = ''LoadBalancer''>
  /opt/LB/scripts/addLBWorker.sh</actionScript>
    <parameters>${LocalIP} ${ajpPort}
    </parameters>
    <placeHolder key = ''ajpPort'' type =
  ''INTEGER''origination = ''Configuration''/>
    <placeHolder key = ''LocalIP'' type =
  ''IPADRESS'' origination = ''Runtime'' scope =
  ''CLUSTER''/>
</scaleOutAction>
```

When scaling actions are to be executed, the "ActionExecutor" (see Figure 2) will first collect the list

Listing 4 Just-in-time resolution of placeholders.

```
def: justInTimeResolution(cluster, clusterMember, placeholder):
    if placeholder.isConfiguration:
            placeholderValue=retrieveValueFromCache(cluster, clusterMember, placeholder.key)
            if placeholderValue is None: # this is the first time to resolve this placeholder
                # retrieve users' configuration that is recorded in the "RuntimeDeployment" repository
                # the value will also be cached to improve performance
                # the cached value will be invalidated when users update the configurations
                placeholderValue=retrieveValueFromRuntimeDeployment(placeholder.key)

else: #it's a placeholder from "Runtime"
        # turns to the "SystemObserver" for the latest value
        If placeholder.scope==="Cluster":
                placeholderValue=SystemObserver.getRuntime(cluster, placeholder.key)
        else:
                placeholderValue=SystemObserver.getRuntime(clusterMember, placeholder.key)
        # Runtime values will not be cached
# validate the value before return
if placeholder.validateValue(placeholderValue):
        return placeholderValue
else:
        reportError(placeholder, placeholderValue)
```

of placeholders. Each placeholder will be resolved, as **Listing 4** shows.

Evaluation

The EAF has been fully implemented, and it has been successfully deployed and verified in an IBM-internal cloud-computing environment [13] and several customer sites. The types of applications that are used to verify the EAF include web applications running in Tomcat clusters and WebSphere application server clusters, Web 2.0 applications hosted by sMash clusters, and map-reduce applications deployed in Hadoop** clusters.

We evaluated the EAF with respect to two aspects. One evaluation relates to ensuring that the applications with different scaling policies can properly scale under different workloads and interferences. The other aspect involves verifying the self-management capability and self-scalability of the EAF as it can be scaled as the number of registered application clusters increases.

To evaluate the first aspect, we provisioned three clusters, one load-balancer cluster and two application-server clusters (named A and B). The load-balancer cluster is based on nginx (wiki.nginx.org), which is a lightweight and high-performance web reverse proxy. Both application server clusters are based on Tomcat (tomcat.apache.org), which is an open-source web container. Initially, there is only one load-balancer instance for the load-balancer cluster, two Tomcat instances for cluster A, and one Tomcat instance for cluster B. Each instance in the system (nginx or Tomcat) is manifested as preinstalled packages in a Xen VM involving a 1-GB disk, 512-MB memory, and 1-GHz CPU. The scaling rules for these clusters are defined in **Table 1**. Clusters A and B share the load-balancer cluster. Once the scale-out rules are satisfied, a new Tomcat or load-balancer instance will be provisioned and launched. Once the scale-in rules are satisfied, an existing Tomcat or load-balancer instance will be killed.

We perform the evaluation tests with two kinds of jobs, namely, CPU-intensive jobs that perform complex arithmetic operations and memory-intensive jobs that continuously create large objects for a period of time. The two jobs constantly send requests to the load-balancer cluster, which further dispatches requests to back-end Tomcat clusters.

Figure 4(a) shows the workload generated by two different jobs. **Figure 4(b)** shows the cluster size changes according to scaling-in and scaling-out actions. In Figure 4(b), we can observe that after running for a period of time, as more requests entered the system, the scale-out rule of cluster A was satisfied at first. As the cluster-scaling rule of B is stricter than that of cluster A, it needs more time to meet the scaling criteria. In addition,

Table 1 Scaling rules for Tomcat cluster A, Tomcat cluster B, and load-balancer cluster (Avg: average; mem: memory; no.: number of).

Scaling rules	*Tomcat cluster A*	*Tomcat cluster B*	*Load-balancer cluster*
Scale-out rule	Avg. CPU utilization > 60% *or* Avg. mem. consumption > 60%	Avg. CPU utilization > 60% *and* Avg. mem. consumption > 60%	(no. of Tomcat servers) > (3 × no. of load balancers)
Scale-in rule	Avg. CPU utilization < 20% *or* Avg. mem. consumption < 20%	Avg. CPU utilization < 20% *and* Avg. mem. consumption < 20%	(no. of Tomcat servers) < (2 × no. of load balancers)

as more Tomcat instances are added, the load-balancer cluster also scales out.

The coordination of scaling actions is necessary to guarantee the correctness of the whole system. However, coordination becomes more complicated in larger scale multitier distributed systems. Most existing industry solutions ([5, 9, 10]) delegate such coordination to application developers. In that case, application developers must implement similar functionalities relating to scaling-action coordination. More importantly, it may become difficult for application developers to coordinate shared clusters (e.g., a shared load-balancing service and a shared database service) that are beyond their control. In EAF, coordination is automated to facilitate autoscaling of different clusters; application developers can focus on the implementation of cluster-specific actions.

To verify EAF self-scalability, a simulation-based approach is adopted. The EAF is hosted in a VM with 4 GB of memory and a 2.16-GHz CPU. The scalable clusters are all running in another VM with 8 GB of memory and a 2.16-GHz CPU. We use long-running threads to simulate the clusters. Every second, a new thread (cluster) is created and starts running. Each one is bound to a workload model (constant, random, or periodic) with one of three metrics, namely, CPU utilization, memory consumption, or response time. Each model also has a predefined scaling threshold (see **Table 2**) and actions. In our experiments, the action is to let the thread sleep for some time between 1 and 10 seconds to simulate the time needed for scaling-out or scaling-in actions.

After being created, each cluster is registered to the EAF. The EAF itself is the first one to be registered, thus makes it self-managed. Every 10 seconds, the EAF initiates one independent thread for each cluster to verify the scaling policy if the cluster is not scaling at that time. Once the time used to verify the scaling policy of a cluster exceeds one second, EAF will take the scale-out actions using the scaling-out rule we set for the EAF in this experiment. In the real world, users could adjust the threshold according to their requirements.

Figure 4(c) shows the maximum time (from start to "now") used by the EAF to verify scaling policies. In the figure, we can observe that, as increasingly more clusters are created and registered, the workload of EAF increases, along with the verification time. When there are 1,401 clusters (among them, 66 clusters start scaling out and 139 clusters start scaling in), the maximum verification time exceeds the threshold (1.015 seconds > 1 second). Therefore, the EAF scales out; a new EAF instance is added. After that, the average workload of the EAF cluster drops and the scaling policy verification time also drops ("now" is the average maximum time for the two EAF instances). When there are 2,848 clusters, the threshold is reached again (1.034 seconds > 1 second), and the EAF cluster scales out for the second time.

From the above evaluation results, we can see that the EAF provides a powerful self-managed mechanism for managing large volumes of scalable clusters without an ever-growing policy verification time. As far as we know, no similar self-management capability is found in existing solutions.

Related work

Cloud computing and dynamic resource provisioning has been recently a popular topic in both industry and academic fields. Most cloud-computing providers offer autoscaling capability with different focuses.

The autoscaling feature of Amazon Web Service (AWS) allows users to scale Amazon EC2** capacity automatically up or down according to predefined scaling rules [5], which are based on performance metrics such as CPU utilization, network activity, or disk utilization. The most important difference between our work and the work of Amazon is that AWS autoscaling works on an infrastructure layer without considering the deployed application characteristics and special requirements. The applications cannot integrate their own scaling policies into the system,

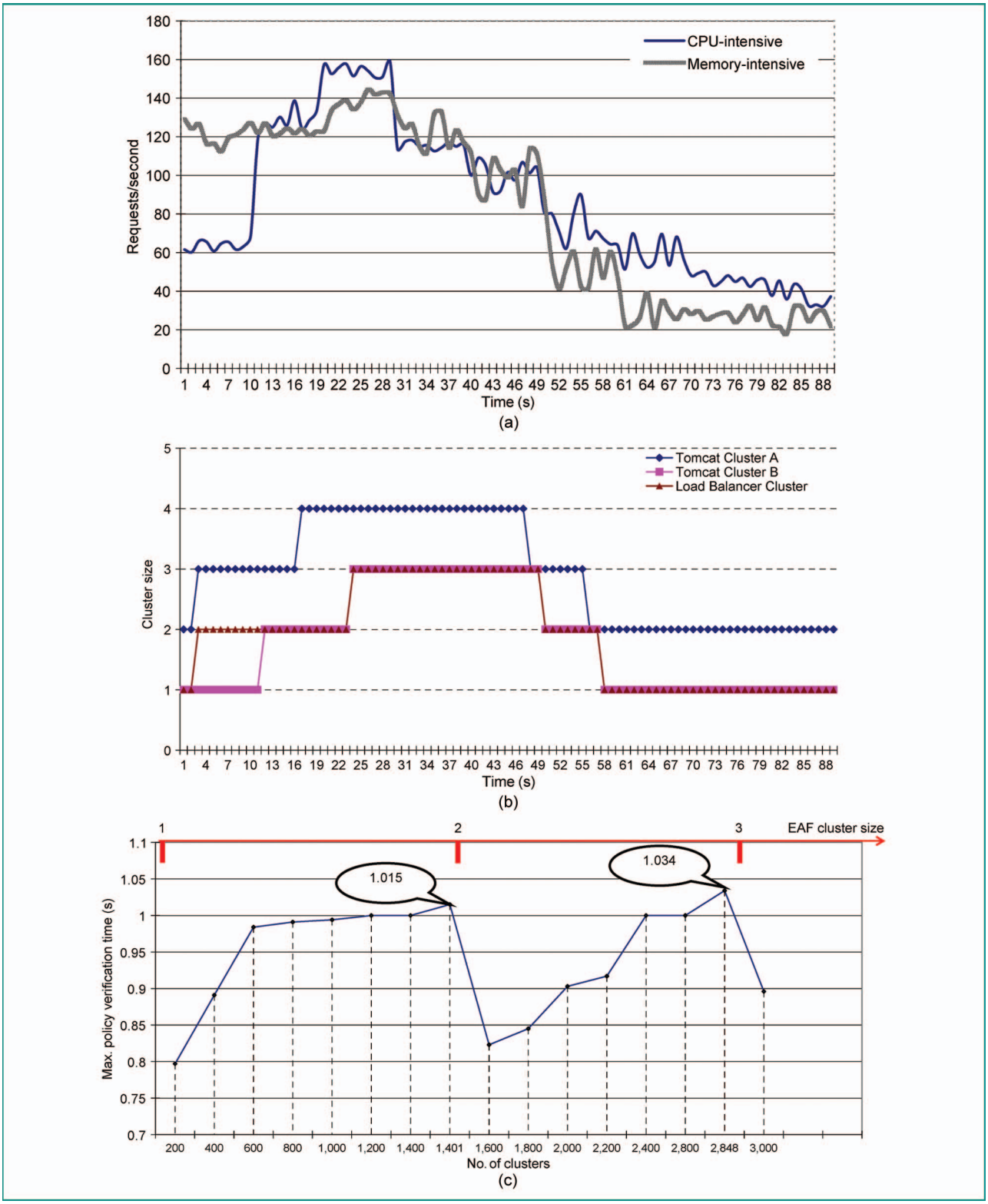

Figure 4

Evaluation of the extreme automation framework. (a) Incoming requests through load balancer. (b) Cluster size according to request change. (c) Scaling of the extreme automation framework.

Table 2 Scaling rules for simulated clusters registered in the extreme automation framework.

	CPU utilization	*Memory consumption*	*Response time*
Scale-out	>90%	>90%	>1,000 ms
Scale-in	<20%	<20%	<100 ms

and application developers must ensure that applications can work in a dynamic infrastructure.

Microsoft Azure** is a platform for running Windows** applications and storing application data in the cloud-computing environment [10]. These applications must be built on the .NET Framework. In Windows Azure, each application has a configuration file. By changing the information in this file manually or programmatically, the owner of an application can scale the application by setting the number of instances that Windows Azure should run. Similarly, application developers need to handle the coordination of scaling actions at their own risk.

Makara [9], the platform-as-a-service solution provider acquired by Red Hat, also provides technologies for organizations to deploy, manage, monitor, and autoscale applications in a cloud-computing environment. However, the scaling policy of Makara is rigid and involves simple triggers and thresholds (currently only supports CPU and requests), and a hardcoded checking interval (10 minutes). These limitations on the autoscaling capability cannot match the diversity of scalable applications in a cloud-computing environment.

RightScale** offers a higher level approach to achieving just-in-time scalability for applications [11]. It provides a design environment that developers can use to setup and manage server clusters on cloud-computing providers such as Amazon EC2. Its predefined templates reduce the time and technology staff needed to set up web applications and websites on cloud-computing platforms. However, at the most fundamental level, RightScale is a configuration and monitoring dashboard for controlling servers and images on cloud-computing platforms. To set up an autoscaling deployment environment, it still requires application developers must ensure that the inputs for templates are properly configured and that applications will correctly boot.

In short, at the time of writing this paper, most existing industry solutions cannot achieve the extreme automation for scaling applications in a cloud-computing environment. Furthermore, the EAF proposed in this paper can be self-managed and scaled during runtime.

In academia, quite a few works address the autoscaling issue in a cloud-computing environment or virtualized systems. Reference [7] presents an autoscaling mechanism that automatically scales up or scales down the underlying cloud-computing infrastructures to adapt to workload change based on application deadline and budget information. It uses an application-level performance metric (i.e., response time) instead of infrastructure-level metrics. The performance model presented in [7] is too simplified to apply it directly to the multitier application architecture. Additionally, the dependencies among clusters are not considered.

Reference [8] proposes a self-scaling framework Tide for virtualized data center management. It makes use of the vSphere management layer of VMware and scales with the management workload. The paper focuses on fast provisioning of virtualized management instances without detailed information of scaling policy and action.

In [14], to avoid overload or underutilizing common resource in a VM host environment, the authors propose what they refer to as a "friendly" VM framework to adaptively scale up or scale down the resource requests of the VM. Similar to our autoscaling framework, the friendly VM framework also delegates the scaling decisions to each VM based on overload detection. However, the friendly VM framework is different from our framework in two important aspects. First, there is no scaling policy definition in friendly VM; instead, its scaling is only based on one metric, i.e., virtual clock time. Second, scaling of a friendly VM is restricted in the VM itself; the friendly VM framework does not take into account the correlation of scaling actions in multiple VMs. Additionally, its scaling actions only apply to two VM-level parameters, namely, the number of active threads in an application and the periodic sleep interval for an application. By contrast, we support the coordination of multiple scaling actions that apply to both VM-level and application-level parameters.

In [15], the authors propose a workload-prediction-based approach to make scaling decision in advance in order to take nonnegligible virtual resources setup time into consideration. This proactive autoscaling is very useful as it can compensate for virtual resource setup time and achieve greater insights than reactive approaches. This proactive scaling approach can be integrated with our casual scaling actions to achieve global optimal autoscaling.

Conclusion

To autoscale applications in a dynamic cloud-computing infrastructure, there are several challenges, including declaration of extensible scaling rules to satisfy application-specific requirements, the coordination of scaling actions that may interfere with each other, and the resolution of dynamic information that can only be determined during runtime. In this paper, an EAF was proposed to facilitate autoscaling of a large volume of clusters through delegated rule verification, centralized action coordination, and just-in-time data resolution. Additionally, the EAF is self-managed and scalable to avoid becoming

the bottleneck. This framework has been implemented and can be broadly applied to different types of applications, including web applications running in Tomcat clusters and WebSphere application server clusters, Web 2.0 applications hosted by sMash clusters, and map-reduce applications deployed in Hadoop clusters. Our experiment has demonstrated its feasibility in the real world.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Facebook, Inc., Apache Software Foundation, Python Software Foundation, Amazon.com, Inc., Microsoft Corporation, or RightScale, Inc., in the United States, other countries, or both.

References

1. T. Hoff, *Friends for Sale Architecture—A 300 Million Page View/Month Facebook RoR App*. [Online]. Available: http://highscalability.com/friends-sale-architecture-300-million-page-view-month-facebook-ror-app
2. *Fabernovel Consulting, Facebook: The* '*Social Media*' *Revolution—A Study and Analysis of the Phenomenon*. [Online]. Available: http://www.fabernovel.com/facebook_en.pdf
3. J. Meattle, *YouTube vs. MySpace Growth: No Contest!* [Online]. Available: http://blog.compete.com/2006/10/18/youtube-vs-myspace-growth-google-charts-metrics/
4. L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *ACM SIGCOMM Comp. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, Jan. 2009.
5. Amazon Inc., *Auto Scaling*. [Online]. Available: http://aws.amazon.com/autoscaling/
6. Google Inc., *Google AppEngine*. [Online]. Available: http://code.google.com/appengine/
7. M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Proc. 11th ACM/IEEE Int. Conf. Grid Comput. (Grid)*, Brussels, Belgium, Oct. 25–28, 2010, pp. 41–48.
8. S. C. Meng, L. Liu, and V. Soundararajan, "Tide: Achieving self-scaling in virtualized datacenter management middleware," in *Proc. 11th Int. Middlew. Conf. Ind. Track*, Bangalore, India, Nov. 2010, pp. 17–22.
9. Red Hat Inc., *Makara Cloud Application Platform*. [Online]. Available: http://www.makara.com/
10. Microsoft Inc., *Windows Azure Platform*. [Online]. Available: http://www.microsoft.com/windowsazure/
11. RightScale Inc., *Web-based Cloud Computing Management Platform by RightScale*. [Online]. Available: http://www.rightscale.com/
12. J. Yang, J. Qiu, and Y. Li, "A profile-based approach to just-in-time scalability for cloud applications," in *Proc. IEEE Int. Conf. Cloud Comput. (CLOUD-II)*, Bangalore, India, Sep. 21–25, 2009, pp. 9–16.
13. K. Ryu, X. L. Zhang, G. Ammons, V. Bala, S. Berger, D. Silva, J. Doran, F. Franco, A. Karve, H. Lee, J. A. Lindeman, A. Mohindra, B. Oesterlin, G. Pacifici, D. Pendarakis, D. Reimer, and M. Sabath, "RC2: A living lab for cloud computing," in *Proc. 24th Int. Conf. Large Install. Syst. Admin. (USENIX LISA)*, Berkeley, CA, Feb. 2010, pp. 201–208.
14. Y. T. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West, "Friendly virtual machines," in *Proc. 1st ACM/USENIX Int. Conf. VEE*, Chicago, IL, Jun. 11/12, 2005, pp. 2–12.
15. E. Caron, F. Desprez, and A. Muresan, "Forecasting for grid and cloud computing on-demand resources based on pattern matching," in *Proc. 2nd IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Indianapolis, IN, Nov. 2010, pp. 456–463.

Received January 31, 2011; accepted for publication March 17, 2011

Jie Yang *Shanda Innovations, Cloud Computing Institute, Block D, Tsing Hua Tongfang Hi-Tech Plaza, Haidian District, Beijing 100083, China (yangjie.jay@snda.com).* Dr. Yang was formerly a Staff Researcher in the Distributed Computing Department at the China Research Lab (CRL), and he currently works at the Cloud Computing Institute of Shanda Innovations. He received the B.S. and Ph.D. degrees in computer science from Peking University, Beijing, China, in 2001 and 2007, respectively. He subsequently joined IBM at the China Research Lab, where he has worked on cloud computing related projects. In 2009, he received an IBM Accomplishment Award for his work on distributed deployment model. He is an author or coauthor of four patents and 12 research papers.

Tao Yu *IBM Research Division, China Research Lab, Haidian District, Beijing 100193, China (yutaoyt@cn.ibm.com).* Dr. Yu is a Research Staff Member in the Distributed Computing department at the China Research Lab (CRL). She received the B.S. degree in computer engineering from University of Electronic Science and Technology of China, Chengdu, China, in 1995, an M.S. degree in electrical engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 1998, and a Ph.D. degree in computer science from University of California, Irvine, in 2006. She joined IBM CRL in 2010, where she has worked on cloud-computing-related projects. Previously, she was a research scientist in HP Labs China. Her research interests include distributed systems, data and information management, and web service and quality-of-service management. She is an author or coauthor of 4 patents and more than 20 research papers in international conferences and journals.

Li Rong Jian *IBM Research Division, China Research Lab, Haidian District, Beijing 100193, China (jianlr@cn.ibm.com).* Mr. Jian is a Researcher in IBM Research Division–China. He received the B.S. and M.Phil. degrees in computer science from Tsinghua University, Beijing, China, in 2008 and Hong Kong University of Science and Technology, Kowloon, Hong Kong, in 2010, respectively. His research interests include distributed computing, cloud computing, pervasive computing, and wireless sensor networks. He is an IEEE member.

Jie Qiu *IBM Research Division, China Research Lab, Haidian District, Beijing 100193, China (qiujie@cn.ibm.com).* Mr. Qiu is a Research Staff Member and Manager in the Distributed Computing and System Management department at the IBM Research Division–China. He received the B.S. and M.S. degrees in computer science from Beijing Institute, Beijing, China, in 2001 and 2004, respectively. He subsequently joined IBM at IBM Research–China, where he has worked on distributed computing. In 2008, he received an IBM Software Accomplishment Award for his contribution to the Rational Deployment Planner product. He is an author or coauthor of 12 patents and 12 technical papers. Mr. Qiu is program committee (PC) co-chair for ACM CloudDB09 and CloudDB10. He also was invited for keynote lectures at the Hadoop Summit China 2010.

Ying Li *IBM Research Division, China Research Lab, Haidian District, Beijing 100193, China (lying@cn.ibm.com).* Dr. Li is a Senior Technical Staff Member and Senior Manager in IBM Research–China. She received the Ph.D. degree in computer science engineering from Northwestern Polytechnical University, Xi'an, China, in 2001. Her major research areas include distributed computing, cloud computing, and service management. She served on the program community of IEEE international conferences including IEEE International Conference on Web Services (ICWS), SCC (IEEE Conference on Services Computing), and EDOC (IEEE Enterprise Distributed Object Computing Conference).