J Supercomput (2013) 65:154–184
DOI 10.1007/s11227-011-0710-5

Peer-to-peer service provisioning in cloud computing environments

Rajiv Ranjan · Liang Zhao

Published online: 21 October 2011

Abstract This paper aims to advance the management and delivery of services in large, heterogeneous, uncertain, and evolving cloud computing environments. The goal is important because such systems are becoming increasingly popular, yet existing service management methods do not scale well, and nor do they perform well under highly unpredictable conditions. If these problems can be solved, then Information Technology (IT) services can be made to operate in more scalable and reliable manner.

In this paper, we present a peer-to-peer approach for managing services in large scale, dynamic, and evolving cloud computing environments. The system components such as virtualized services, computing servers, storage, and databases self-organize themselves using a peer-to-peer networking overlay. Inter-networking system components through peer-to-peer routing and information dissemination structure is essential to avoid the problems of management bottleneck and single point of failure that is predominantly associated with traditional centralized and hierarchical distributed (grids/clouds) system design approaches. We have validated our approach by conducting a set of rigorous performance evaluation study using the Amazon EC2 cloud computing environment. The results prove that managing services based on peer-to-peer routing and information dissemination structure is feasible and offers

R. Ranjan (✉) · L. Zhao
Service Oriented Computing (SOC) Research Group, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia
e-mail: rajivr@cse.unsw.edu.au

L. Zhao
e-mail: lzhao@cse.unsw.edu.au

R. Ranjan
Information Engineering Laboratory, CSIRO Information and Communication Technologies (ICT) Centre, North Road, Acton, ACT 2601, Australia
e-mail: rajiv.ranjan@csiro.au

significant performance benefits as regards to overall system reliability, scalability, and self-management.

Keywords Peer-to-peer · Service provisioning · Cloud computing

1 Introduction

Cloud computing is the latest evolution of computing, where IT capabilities are offered as services. Cloud computing [6, 10, 15, 37] delivers infrastructure, platform, and software (application) as services, which are made available as subscription-based services in a pay-as-you-go model to consumers. These services in industry are respectively referred to as Infrastructure as a Service (IaaS) [38, 40], Platform as a Service (PaaS) [2–4, 32], and Software as a Service (SaaS) [13]. A technical report [6] published by University of Berkeley in February 2009 states that "Cloud computing, the long-held dream of computing as a utility, has the potential to transform a large part of the IT industry, making software even more attractive as a service."

If cloud computing is properly applied within an overall IT strategy, it can help SMEs and governments to lower their IT costs, by taking advantage of economies of scale and automated IT operations, while at the same time optimizing investment in in-house computing infrastructure. Adoption of cloud computing platforms as an application service provisioning environment has the following critical benefits: (i) software enterprises and startups with innovative ideas for new Internet services are no longer required to make large capital outlays in the hardware and software infrastructures to deploy their services or human expense to operate it; (ii) government agencies and financial organizations can use clouds as an effective means for cost cutting by leasing their IT services hosting and maintenance responsibility to external cloud(s); (iii) organizations can more cost effectively manage peak-load by using the cloud, rather than planning and building for peak load, and having under-utilized servers sitting there idle during off peak time, and (iv) failures due to natural disasters or regular system maintenance/outage may be managed more gracefully as services may be more transparently managed and migrated to other available cloud resources, hence enabling improved service level agreement (SLA).

The process of deploying application services on publicly accessible clouds (such as Amazon EC2 [38], Google App Engine [17], Rejila [31], Rackspace [24]) that expose their capabilities as a network of virtualized services (hardware, storage, database) is known as Cloud Provisioning. The Cloud provisioning [26] process consists of three key steps: (i) *Virtual Machine (VM) Provisioning*, this involves instantiation of one or more VMs that match the specific hardware characteristics and software requirements of the services to be hosted. Most cloud providers provide a set of general-purpose VM classes with generic software and resource configurations. For example Amazon's EC2 cloud supports five basic types of VMs; (ii) *Resource Provisioning*, which is mapping and scheduling of VMs onto distributed physical cloud servers within a cloud. Currently, most of cloud providers do not provide any control over resource provisioning to application service-level developers, in other words mapping of VMs to physical servers is completely hidden from cloud application service developers; and (iii) *Service Provisioning*, the final step is deployment of specialized application services (such as web services, business processes, Hadoop/MapReduce processes for scalable data analysis) within VMs and mapping of end-user's requests to these services. In this paper, we mainly focus on the third step, where given a set of VMs that are hosting different types of application services (or multiple instances of same application service), how to dynamically monitor, and distribute the incoming requests among them in a completely peer-to-peer (decentralized and distributed) manner.

Although the components (services, VMs, physical servers, storage) that are part of a cloud provisioning environment may be distributed, existing techniques usually employ centralized approaches [2, 4, 40] to overall service monitoring, discovery, and load-balancing. We claim that centralized approaches are not an appropriate solution [26, 29, 41] for this purpose, due to concerns of scalability, performance, and reliability arising from the management of multiple service queues and the expected large volume of service requests. Monitoring of system components is required for effecting on-line load-balancing through a collection of system performance characteristics. Therefore, we advocate architecting service monitoring, discovery, and load-balancing services based on decentralized (peer-to-peer) messaging and indexing models.

Services provisioned across multiple clouds are located in different network domains that may use heterogeneous addressing and naming schemes (public addresses, private addresses with NAT, etc.). In general, services would require all their distributed components to follow a uniform IP addressing scheme (for example, to be located on the same local network), so it becomes mandatory to build some kind of overlay network on top of the physical routing network that aids the service components in undertaking seamless and robust communication. Existing implementation including VPN-Cubed [39], OpenVPN [25] provides an overlay network that allows application developer to control addressing, topology, protocols, and encrypted communications for services deployed across multiple clouds (private and public) sites. However, these implementations do not provide capabilities related to decentralized service discovery, monitoring, and load-balancing across VM instances.

In this paper, we advocate architecting service monitoring [28], discovery [34], and load-balancing [26] services based on decentralized (peer-to-peer) messaging [9, 18, 22, 23] and indexing [14, 19, 29, 36] models.

1.1 Scalable peer-to-peer approach

In this paper, we propose to interconnect the cloud system components based on structured peer-to-peer routing structures. In literature, structured peer-to-peer routing structures are more commonly referred to as the DHTs. DHTs provide hash table like functionality at Internet scale. DHTs such as Chord [35], CAN [30], and Pastry [33] are inherently self-organizing, fault-tolerant, and scalable. DHTs provide services that are light-weight and hence, do not require an expensive hardware platform for hosting, which is an important requirement as regards to building and managing cloud systems that aggregate massive number of commodity servers and virtualized instances hosted within them. A DHT is a distributed data structure that associates a key with a data. Entries in a DHT are stored as a (key, data) pair. A data can be looked up within a logarithmic overlay routing hops if the corresponding key is known.

Engineering cloud provisioning services based on DHTs is an efficient approach because DHTs are highly scalable, can gracefully adapt to the dynamic system expansion (join) or contraction (leave, failure), are not susceptible to single point of failure, and promote autonomy. Engineering Cloud provisioning services over a DHT networking model offers *new research challenges* in designing novel data indexing and routing algorithms. Innovative techniques need to be developed to ensure that query load is balanced across the system; routing links per resource is minimized, and at the same time logarithmic performance bounds for data lookup, entity (services, resources) join or leave on DHTs are maintained.

1.2 Our contributions

The *novel contributions* of this paper include: (i) an integrated framework called *cloud peer* that facilitates peer-to-peer management of cloud system components for providing end-users with fault-tolerant and reliable services in large, autonomous, and highly dynamic environments; (ii) extension of the DHT routing structure with cloud service monitoring, discovery, and load-balancing capabilities; and (iii) comprehensive asymptotic analysis of messaging overhead involved with proposed approach.

We now summarise some of our findings:

– in a cloud computing system consisting of n services, on average the number of messages required to successfully discover and map a request to a service is $O(\log n)$.
– proposed peer-to-peer approach to load-balancing is highly effective in curbing the number of mapping iterations undertaken on a per service request basis, the mapping and notification message complexity involved with the load-balancing algorithm is $O(1)$.
– application workload granularity and service status update frequency have significant influence on the mapping delay experienced by service requests in the system.

1.3 Paper organization

The rest of this paper is organized as follows: First, existing state-of-the-art in cloud provisioning domain is discussed. This is followed by some survey results on cloud provisioning capabilities in leading commercial public clouds. Then, a layered approach to architecting peer-to-peer cloud provisioning system is presented. The finer details related to architecting peer-to-peer cloud service discovery and load-balancing techniques over DHT overlay is then presented, followed by a analysis of the algorithms. Lastly, experimental results of the peer-to-peer cloud provisioning implementation across a public cloud (Amazon EC2) environment is presented next. The paper ends with brief conclusive remarks.

2 Related work

In this section, we look at the current state-of-the-art and compare it against the work proposed in this paper. Key players in cloud computing domain such as Amazon EC2 [38], Microsoft Azure [40], Google App Engine [17], Eucalyptus [12], and GoGrid [16] offer a variety of pre-packaged services for monitoring, managing and provisioning resources. However, the way these techniques are implemented in each of these clouds vary significantly.

Currently, Amazon Web Services (AWS) exposes three *centralized* services for load-balancing (Elastic Load Balancer [4]), monitoring (Amazon CloudWatch [2]), and Auto-Scaling (Amazon Auto-Scaling [1]). Both Elastic Load Balancer and Auto-Scaling services rely on the resource status information reported by the CloudWatch service. Elastic Load Balancer service can automatically provision incoming service workload across available Amazon EC2 instances. On the other hand, Auto Scaling service can be used to dynamically scale-in or scale-out the number of Amazon EC2 VM instances for handling changes in service demand patterns. However, CloudWatch can only monitor the status information at VM-level not at specific application service level. In reality a VM instance can host more than one services such as web server, database backend, image server, etc. Therefore, there is critical requirement monitoring the status of individual services in a scalable manner. This can aid in accurately predicting services' behaviours and performance.

Eucalyptus [12] is an open source cloud computing platform. The system is composed of three controllers such as Node Controller, Cluster Controller, and Cloud Controller. These controllers are responsible for managing virtual machines on physical resources, coordinating load-decisions across nodes that are in same availability zone and handling connections from external clients and administrators. In the current hierarchical design, cloud controller works at the root level, Cluster Controllers are the intermediate nodes, and Node Controllers operate at the leaf level. However, from network management perspective the *hierarchical* design pattern may prove to be performance bottleneck with the increase in service request intensity and system size.

Windows Azure Fabric [40] is designed over a weave-like inter-connection structure that includes nodes (servers, VMs and load-balancers) and edges (power, Ethernet and serial communications). The Azure Fabric Controller (FC) is the service, which monitors, maintains and provisions machines to host the applications that the developer creates and deploys in the Microsoft Cloud. FC is responsible for managing all the software and hardware components in a Microsoft data center. These components include servers, load-balancers (hardware-based), switches, routers, power-on automation devices, etc. The behavior of the FC is made redundant by creating multiple replicas (5 to 7) at any given point of time. These replicas are constantly updated to ensure that information consistency and integrity is achieved across FCs.

GoGrid [16] Cloud Hosting offers developers centralized F5 Load Balancers for distributing application service traffic across servers, as long as IPs and specific ports of these servers are attached. The load balancer allows Round Robin algorithm and Least Connect algorithm for routing application service requests. Also, the load balancer can detect a crash of the server, which it handles by redirecting future requests for the failed server to other available servers.

Unlike other cloud platforms, Google App Engine [17] offers developers a scalable platform in which applications can run, rather than providing direct access to a customized virtual machine. Therefore, access to the underlying operating system is restricted in App Engine. And load-balancing strategies, service provisioning and auto scaling are all automatically managed by the system behind the scenes. At this time, there is very little or no documentation available about finer details of Google App Engine architecture.

In this paper, we address the limitation of nonscalable (centralized, hierarchical) way of provisioning services in clouds through implementation of a generic peer-to-peer routing and information indexing structure (*Cloud Peer*) for internetworking multiple services and applications as part of single, cohesive cloud resource management abstraction.

3 Models

This focus of this section is to provide comprehensive details on cloud computing environment. The system and application models considered in this paper are described first, and the proposed peer-to-peer (discovery, load-balancing) model that delivers reliable and scalable application management environment is induced and presented based on these models. Table 1 shows the notations for System, Application, and Index models that we refer to in rest of this paper.

3.1 Overall system model

Clouds aim to power the next generation data centers by architecting them as a network of virtual services (hardware, database, user-interface, application logic) so that end-users are able to access and deploy applications from anywhere in the world on demand at competitive costs depending on users QoS (Quality of Service) requirements. The cloud computing system [21], P, is a set of cloud computing infrastructures owned and maintained by 3rd party providers such as Amazon EC2, Flexiscale, and Go-Grid. More formally,

$$P = (r_1, r_2, \ldots, r_n) \cup d,$$

where

$$r_i, 1 \le i \le k, = \{v_{1,m}, v_{i,2}, \ldots, v_{i,m}\} \cup d_i.$$

In case of a virtual cloud (r_i), v_i is virtual machine for hosting services and d is the cloud specific data repository (such as $S3$ in case of Amazon S3 cloud). In reality, a cloud application provisioning system can create multiple virtual clouds within single or multiple cloud computing environments. The granularity and size (number of virtual clouds) of the system is dependent on the application's QoS requirements. All virtual clouds are fully interconnected in way that a messaging route exists between any two individual VMs. Virtual clouds that are hosted in different cloud computing domains (data centers) are connected to each other through Internet; this implies that inter virtual cloud communication is dynamic and heterogeneous. However, we

Table 1 Notations: system, application, and index models

Symbol	Meaning
System	
P	a set of cloud computing infrastructures
r_i	i-th cloud resource
s_i	a service instance
d_i	data repository for i-th cloud
$v_{i,m}$	a virtual machine instance
Application	
G	an application workload
n	number of tasks or work units in a application
w_i	i-th task or work unit
$I_{d,i}$	input dataset associated with i-th work unit
Multi-dimensional Index	
di, j	discovery query for service i from j-th application provisioner
u_i	a update query issued by the i-th service
dim	dimensionality or number of attributes in the Cartesian space
$f_{\min}$	minimum division level of d-dimensional index tree
x, y, z	Cartesian coordinates of a 3-dimensional index cell
b	a pastry overlay configuration parameter
key	a unique identifier assigned to a query
$T_{d_{i,j}}, T_{u_i}$	random variables denoting number of disjoint query paths undertaken in mapping a discovery and update queries
$M_{d_{i,j}}$	random variable denoting number of messages generating a discovery query

can safely assume that communication network between virtual clouds hosted within same data center would be homogeneous. The performance of cloud resources (such as processor speed, memory, cache size, disk, and storage) that are allocated to a virtual machine can be approximated by measuring capacity, spindle count, seek time, and transfer speed. These software and hardware characteristics determine the performance that a given application can extract from the system.

Each VM can host a service s_i (or services) that provides a certain kind of action, or functionality that's exchanged for value between provider and end-user. The action or functionality can vary based on the application model, for example a *high-performance computing service* such as Folding@home, SETI@home provides functionality for executing mathematical models on given set of data. On the other hand, a Web server is a service that delivers content, such as web pages, using the Hypertext Transfer Protocol (HTTP), over the Internet.

3.2 Application model

Several fundamental workloads in science, physics, commerce, media, and engineering can be modeled as *embarrassingly parallel* or *Bag of Tasks (BoT)* workloads. Some popular examples in this domain are referred to as the "@home" workload (e.g., Einstein@home, Folding@home, SETI@home, BLAST, MCell, INS2D).

A workload G that belongs to this model includes of a number of n homogeneous or heterogeneous independent work units $\{w_1, w_2, \ldots, w_n\}$ without interwork unit control or communication dependencies. A work unit w_i in G is always associated with a set $I_{d,i}$ of input data objects $\{I_{d,1}, I_{d,2}, \ldots, I_{d,n}\}$. In general each work unit represents a particular data analysis or processing experiment with a distinct set of parameter and input data object. Although it can be assumed that there exists a random relationship between data objects and work units, the relationship between data objects and work units may show regular dependency patterns, such as one-to-many and partitioned.

Traditionally in Grid computing domain, an application workload is characterized as computation or communication intensive. The main application characteristic that determines its type is the communication-to-computation ratio. In this work, we assume that work units are computation intensive and they do not communicate at runtime. Thus, there exist no control or data dependency among the work units in the application. In the proposed system model, a SME or Cloud application owner can simultaneously submit multiple applications that are composed under BoT model.

4 Layered system design

Figure 1 illustrates a layered cloud computing architecture for enabling peer-to-peer provisioning of services. Physical cloud servers, along with core middleware capabilities (VMs, hypervisors), form the basis for delivering IaaS. The application-level middleware services aims at providing PaaS capabilities. The top layer focuses on

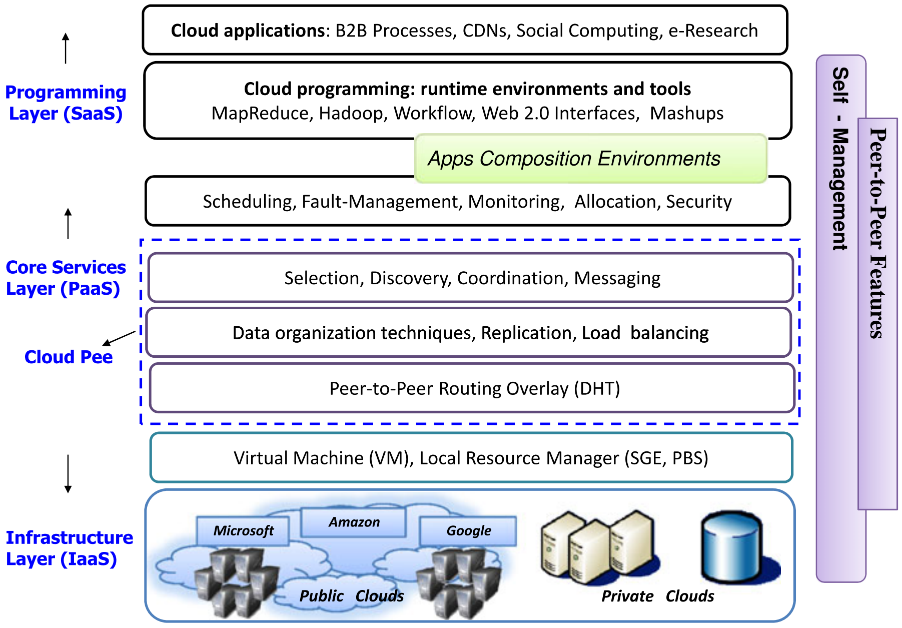

Cloud applications: B2B Processes, CDNs, Social Computing, e-Research
Programming Layer (SaaS)
Cloud programming: runtime environments and tools
MapReduce, Hadoop, Workflow, Web 2.0 Interfaces, Mashups
Apps Composition Environments
Scheduling, Fault-Management, Monitoring, Allocation, Security
Core Services Layer (PaaS)
Selection, Discovery, Coordination, Messaging
Cloud Pee
Data organization techniques, Replication, Load balancing
Peer-to-Peer Routing Overlay (DHT)
Virtual Machine (VM), Local Resource Manager (SGE, PBS)
Infrastructure Layer (IaaS)
Microsoft
Amazon
Google
Public Clouds
Private Clouds
Self - Management
Peer-to-Peer Features

Fig. 1 A depiction of layered peer-to-peer provisioning system architecture

application services (SaaS) by making use of services provided by the lower layers. PaaS/SaaS services are often developed and provided by the 3rd party service providers, who may or may not be different from IaaS providers.

Cloud applications (SaaS): Popular cloud applications include Business to Business (B2B) applications, traditional eCommerce type of applications, enterprise business applications such as CRM and ERP, social computing such as Facebook and MySpace, and compute, data intensive applications and Content Delivery Networks (CDNs). This layer also includes the software environments and programming frameworks such as Web 2.0 Interfaces (Ajax, IBM Workplace, and Visual Studio.net Azure plug-in) that help developers in creating rich, cost-effective, user-interfaces for browser-based applications. The layer also provides the data intensive, parallel programming environments (such as MapReduce, Hadoop, Dryad) and composition tools that ease the creation, deployment, and execution of applications in clouds.

Core Services Layer (PaaS): This layer implements the platform level services that provide runtime environment-enabling cloud computing capabilities to application services built using User-Level Middlewares. Core services at this layer include Scheduling, Fault-Management, Monitoring, Dynamic SLA Management, Accounting, Billing, and Pricing. Further, the services at this layer must be able to provide support for decentralized load-balancing, scalable selection, and messaging between distributed cloud components. Some of the existing services operating at this layer are Amazon EC2's CloudWatch and Load-balancer service, Google App Engine, Microsoft Azure fabric controller, and Aneka. To be able to provide support for decentralized service discovery [20] and load-balancing between cloud components (VM instances, application services); novel Distributed Hash Table (DHT)-based PaaS layer services need to be developed at this layer for supporting complex interactions with guarantees on dynamic management. In Fig. 1, this component of PaaS layer is shown as cloud peer service.

Infrastructure Layer (IaaS): The computing power in cloud computing environments is supplied by a collection of data centers that are typically installed with many thousands of servers. At the IaaS layer there exists massive physical servers (storage servers and application servers) that power the data centers. These servers are transparently managed by the higher level virtualization services and toolkits that allow sharing of their capacity among virtual instances of servers. These virtual machines (VMs) are isolated from each other, which aids in achieving fault tolerant behavior and the isolation of security contexts.

5 The peer-to-peer provisioning approach

In this paper, we consider a service provisioning approach that involves following stages: (i) *Discovery*, the process of locating an appropriate application service in a loosely-networked overlay of services that can successfully serve an user's request under given constraints/targets (type, availability, location, performance constraints) and (ii) *Load-balancing*, which is an act of uniformly distributing workload across one or more service instances, in order to achieve performance targets such as maximize resource utilization, maximize throughput, minimize response time, minimize cost, and maximize revenue. Although resource discovery and load-balancing are widely studied problems in distributed systems (grid computing, data centers), more advanced techniques and algorithms that cater for higher level of decentralization, scalability, and self-management need to be developed.

In effect, end-users for an application service could initiate request from any part of the Internet (in other words end-users are geographically and topologically distributed). Similarly, the system size can vary based on popularity of application service and performance requirements. Further, as clouds become ready for mainstream acceptance, scalability of services will come under more severe scrutiny since at that time cloud providers will have to support an increasing number of online services, each being accessed by massive numbers of global users round the clock. Traditional way of discovering and mapping requests (stage 2) across services based on centralized network models is inefficient due to scalability, performance, and reliability concerns arising from large system size and volume of service requests. Thus, scalable and decentralized approaches that are self-managing must be developed to accomplish tasks at different stages (stage 1 & 2).

5.1 Query types and their composition

To an extent, the multilayered (IaaS, PaaS, and SaaS) architecture of cloud computing environments (refer to Fig. 2) complicates the overall service discovery problem. At each layer, cloud offers heterogeneous mix of services that differ in their scale, granularity, and type. At IaaS layer, cloud offers services such as physical servers, VMs, storage devices. The search dimensions at IaaS layer can include processor speed, number of cores, processor architecture, installed operating system, available memory, and network bandwidth. At SaaS-level, SaaS providers can effectively deploy multiple types of services (and/or multiple instance of same service) based on the composition and predicted workload of the application. For instance, a SaaS provider, who hosts a business workflow (such as ERP, CRM) can simultaneously create multiple instances of different services (such as application server, database server, security server, etc.) to meet its workload demands. The main challenge for a Service Aggregation Engine (Service Provisioner) is to appropriately select the set of services across workflow such that end-to-end performance targets are met (e.g., minimize response, maximize service throughput).

A query at SaaS-level must search for services that satisfy multiple criteria including Service type, price, hosting location, and availability. For example, based on recent information published by Amazon EC2 CloudWatch service, each Amazon Machine Image (AMI) instance has seven performance metrics (see Table 2) and three dimensions (see Table 3) associated with it. To summarize, a service discovery query would be a conjunction of sub-queries that needs to be resolved across multiple search dimensions of IaaS and Saas. An example of such a query follows:

Cloud Service Type = web hosting && Host CPU Utilization $\leq$ 50% && Instance OSType = WinSrv2003 && Host Processor Cores $\geq$ 1 && Host Processors Speed $\geq$ 1.5 GHz && Host Cloud Location = Europe

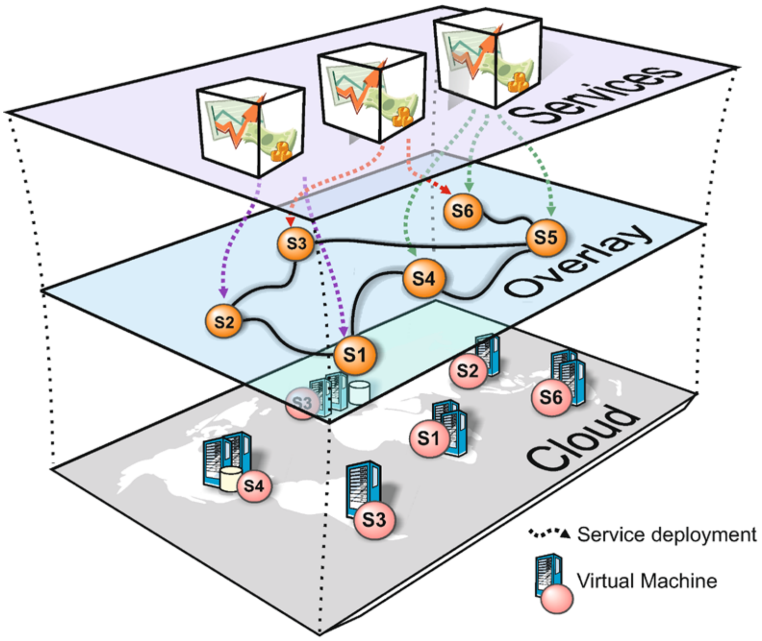


Fig. 2 Visual representation of mapping SaaS to logical PaaS-level overlay and IaaS-level physical Cloud servers

Table 2 Performance metrics associated with an Amazon EC2 AMI instance

Utilization	IncomingTraffic	OutgoingTraffic	DiskWriteOps
DiskReadBytes	DiskReadOps	DiskWriteBytes	

Table 3 Performance dimensions associated with an Amazon EC2 AMI instance

Image-ID	AutoScalingGroupName	InstanceID	InstanceType

On the other hand, service instances deployed on physical Cloud services needs to publish their information so that service provisioner can search them. Service instances update their software and hardware configuration and the availability status by sending update query. The service configuration distribution in three dimensions is shown in Fig. 3. A service update query has the following semantics:

Cloud Service Type = web hosting && Host CPU Utilization = 30% && Instance OSType = WinSrv2003 && Host Processor Cores = 2 && Host Processors Speed = 1.5 GHz && Host Cloud Location = Europe

5.2 Cloud peer service design

The cloud peer implements services for enabling decentralized and distributed discovery supporting status lookups and updates across the internetworked cloud com-

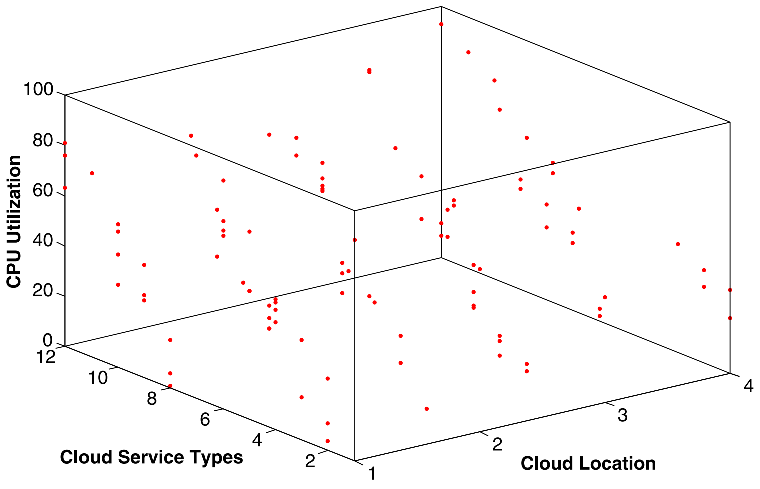


Fig. 3 SaaS Services types and Update query distribution

puting system components; enabling optimizing load-balancing and tackling the distributed service contention problem. Dotted box in Fig. 1 shows the layered design of cloud peer service over DHT based self-organizing routing structure. The services build upon the DHT routing structure extends (both algorithmically and programmatically) the fundamental properties related to DHTs including deterministic lookup, scalable routing, and decentralized network management. The cloud peer service is divided into a number of sublayers (see Fig. 1): (i) higher level services for discovery, coordination, and messaging; (ii) low level distributed indexing and data organization techniques, replication algorithms, and query load-balancing techniques; (iii) DHT-based self-organizing routing structure. A cloud peer undertakes the following critical tasks that are important for proper functioning of DHT-based provisioning overlay:

5.2.1 Overlay construction

The overlay construction refers to how cloud peers are logically connected over the physical network. The system design approach utilizes FreePastry [5] (a open source implementation of Pastry DHT) as the basis for creation of cloud peer overlay. A Pastry overlay inter-connects the cloud peer services based on a ring topology. Inherent to the construction of a Pastry overlay are the following issues: (i) Generation of cloud peer ids and query (discovery, update) ids, called keys, using cryptographic/randomizing hash functions [7, 20, 27] such as SHA-1. These ids are generated from 160-bit unique identifier space. The id is used to indicate a cloud peers position in a circular id space, which ranges from 0 to 2^{160-1}. The queries and cloud peers are mapped on the overlay network depending on their key values.

Each cloud peer is assigned responsibility for managing a small number of queries and building up routing information (leaf set, routing table, and neighborhood set) at various cloud peers in the network. Given the Key k, Pastry routing algorithm can find the cloud peer responsible for this key in $O(\log_b n)$ messages, where b is the base and

n is the number of cloud peers in the network. Each cloud peer in the Pastry overlay maintains a routing table, leaf set, and neighborhood set. These tables are constructed when a cloud peer joins the overlay, and it is periodically updated to take into account any new joins, leaves, or failures. Each entry in the routing table contains the IP address of one of the potentially many cloud peers whose id have the appropriate prefix; in practice, a cloud peer is chosen, which is close to the current peer, according the proximity metric. Figure 4 shows a hypothetical Pastry overlay with keys and cloud peers distributed on the circular ring based on their cryptographically generated ids.

5.2.2 Multi-dimensional query indexing

A 1-dimensional hashing provided by a standard DHT is insufficient to manage complex objects such as resource tickets and claims. DHTs [8] generally hash a given unique value (e.g., a file name) to a 1-dimensional DHT key space and hence they cannot directly support mapping and lookups for complex objects. Management of those queries whose extents lie in d-dimensional space can be done by embedding a logical index structure over the 1-dimensional DHT key space.

In order to support multidimensional query indexing (service type, host utilization, VM instance OS type, host cloud location, host processor speed) over Pastry overlay, a cloud peer implements a distributed indexing algorithm, which is a variant of peer-to-peer MX-CIF Quad tree distributed indexing algorithm [36]. The distributed indexing algorithm builds a multi-dimensional index based on the cloud service attributes, where each attribute represents a single dimension. An example 2-dimensional attribute space that indexes service attributes including Speed and CPU Type is shown in Fig. 4.

First step in initializing the distributed index is the process called Minimum Division ($f_{\min}$). This process divides the attribute space into multiple index cells when the multi-dimensional distributed index is first created. As a result of this process, the attribute space resembles a grid like structure consisting of multiple index cells. The cells resulting from this process remain constant throughout the life of the indexing domain and serve as entry points for subsequent service discovery and update query mapping. The number of cells produced at the minimum division level is always equal to $(f_{\min})^{dim}$, where dim is dimensionality of the attribute space. Every cloud peer in the network has basic information about the attribute space coordinate values, dimensions and minimum division level. Cloud peers can obtain this information (cells at minimum division level, control points) in a configuration file from the bootstrap peer. Each index cell at $f_{\min}$ is uniquely identified by its centroid, termed as the control point. In Fig. 4, $f_{\min} = 1$, $dim = 2$. The Pastry overlay hashing method ($DHT(coordinates)$) is used to map these control points so that the responsibility for an index cell is associated with a cloud peer in the overlay. For example in Fig. 4, $DHT(x_1, y_1) = k10$ is the location of the control point $A(x_1, y_1)$ on the overlay, which is managed by cloud peer 12.

5.2.3 Multi-dimensional query routing

This action involves the identification of the index cells at minimum division level $f_{\min}$ in the attribute space to map a service discovery and update query. For mapping

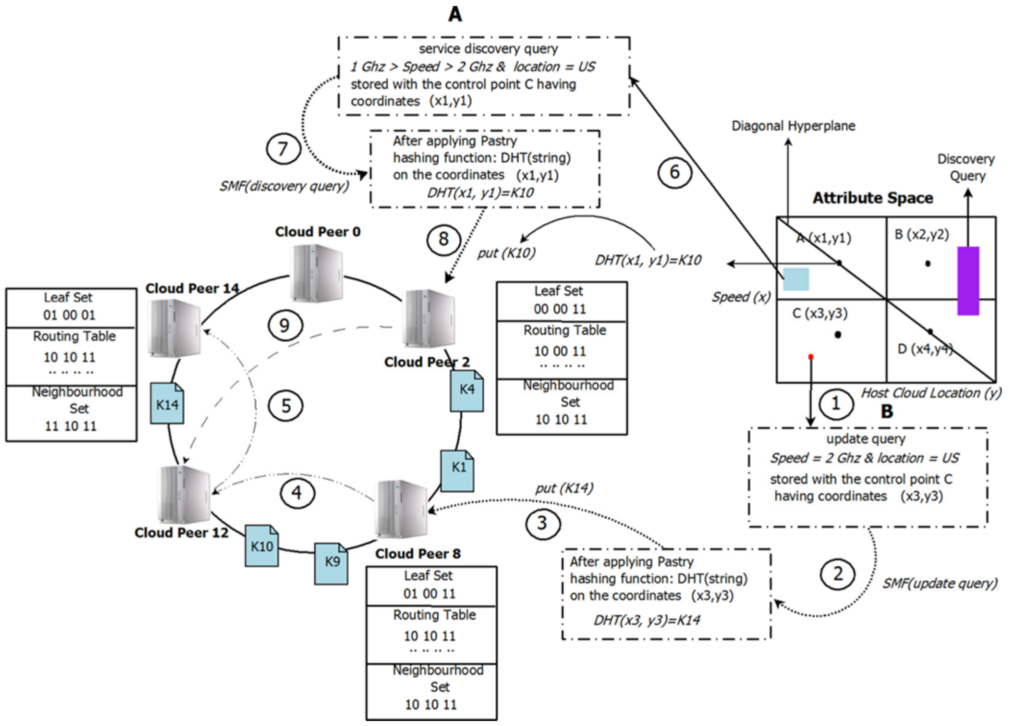


Fig. 4 A pictorial representation of Pastry (DHT) overlay construction, multidimensional data indexing, and routing: (*1*) a service hosted within a VM publishes a update query; (*2*) Cloud peer 8 computes the index cell, $C(x_3, y_3)$, to which the update query maps by using mapping function $SMF(query)$; (*3*) Next, distributed hashing function, $DHT(x_3, y_3)$, is applied on the cells coordinate values, which yields a overlay key, $K14$; (*4*) Cloud peer 8 based on its routing table entry forwards the request to peer 12; (*5*) Similarly, peer 12 on the overlay forwards the request to cloud peer 14; (*6*) a provisioning service submits a service discovery query; (*7*) Cloud peer 2 computes the index cell, $C(x_1, y_1)$, to which the service discovery query maps; (*8*) $DHT(x_1, y_1)$ is applied that yields an overlay key, $K10$; (*9*) Cloud peer 2 based on its routing table entry forwards the mapping request to peer 12

Algorithm 1: Query routing algorithm undertaken by cloud peer service

```
0.1  Input        : discovery $d_{i,j}$, update $u_i$
0.2  Output       : mapping of objects to cloud peers
0.3  Definitions  :
       – SMF (query): returns set C of base index cells to
         which a query maps;
       – DHT (control point): returns the key which acts as
         the basis of mapping query to Pastry overlay;
       – C: set that stores the coordinates for base index cells
         to which a query is mapped to;
       – key: an m-bit identifier generated from $2^m$ space, forms the basis for determin-
         istic routing;
       – put: sends the messages to appropriate peer using Pastry method. This
         method is exposed by FreePastry framework;
     Procedures   :
0.4  Post ()      : map query
0.5  Input        : query $\in$ { discovery, update }
0.6  call Route ($d_{i,j}$);
0.7  return ;

0.8  Route ()     : route query
0.9  Input        : query $\in$ { discovery, update }
0.10 Intialization: C $\leftarrow$ { $\phi$ }, key $\leftarrow$ null
0.11 C = SMF(query);
0.12 foreach {x,y,z } $\in$ C do
0.13     key = DHT ({ x,y,z });
0.14     call put (key);
0.15 end
0.16 return ;
```

service discovery query, the mapping strategy depends on whether it is a multidimensional point query (equality constraints on all search attribute values) or multidimensional range query. For a multi-dimensional point service discovery query the mapping is straight forward since every point is mapped to only one cell in the attribute space. For a multidimensional, mapping is not always singular because a range look-up can cross more than one index cell. To avoid mapping a multidimensional service discovery query to all the cells that it crosses (which can create many unnecessary duplicates) a mapping strategy based on diagonal hyperplane of the attribute space is utilized.

Algorithm 1 shows pseudo code for query routing. This mapping involves feeding the service discovery query's spatial dimensions into a mapping function, *SMF*(*query*) (see line 0.11 in Algorithm 1). This function returns the IDs of index cells to which given query can be mapped (refer to step 7 in Fig. 4. Distributed hashing ($DHT(coordinates)$) is performed on these IDs (which returns keys for Pastry overlay) to identify the current cloud peers responsible for managing the given keys. A cloud peer service uses the index cell(s) currently assigned to it and a set of known base index cells obtained at the initialization as the candidate index cells. Similarly, mapping of update query also involves the identification of the cell in the attribute space using the same algorithm. A update query is always associated with an event region and all cells that fall fully or partially within the event region would be selected to receive the corresponding objects. The calculation of an event region is also based upon the diagonal hyperplane of the attribute space. Giving in depth information here is out of the scope for this paper, however the readers who would like to have more information can refer the paper that describes the index in detail.

5.3 Load-balancing algorithm description

A load-balanced provisioning of requests between virtual machine instances deployed in clouds is critical, as it prevents the service provisioner from congesting the particular set of VMs and network links, which arises due to lack of complete global knowledge. In addition, it significantly improves the cloud user Quality of Service (QoS) satisfaction in terms of response time. The cloud peer service in conjunction with the Pastry overlay and multi-dimensional indexing technique is able to perform a load-balanced service provisioning. The description of the actual load-balancing mechanism follows next.

As mentioned in previous section, both service discovery (issued by service provisioner) and update query (published by VMs or Services hosted within VMs) queries are spatially hashed to an index cell i in the multidimensional attribute space. In Fig. 5, service discovery query for provisioning request P1 is mapped to index cell with control point value A, while for P2, P3, and P4, the responsible cell has control point value C. Note that these service discovery queries are posted by service provisioners. In Fig. 5, a provisioning service inserts a service discovery query with cloud peer p, which is mapped to index cell i. The index cell i is spatially hashed through routing functions (refer to Algorithm 1) to an cloud peer s. In this case, cloud peer s is responsible for balancing the load offered by the service discovery queries that are currently mapped to the cell i. Subsequently, service s_i hosted within VM u issues a update query (see Fig. 5) that falls under a region of the attribute space currently required by the requests P3 and P4. Next, the cloud peer s has to decide which of the requests (either P3 or P4 or both) is allowed to claim the update query published by s_i. The load-balancing decision is based on the principle that it should not lead to over-provisioning of the concerned service. This mechanism leads to load-balanced allocation of work units across services in clouds and aids in achieving system-wide objective function.

The examples in Table 4 are list of service discovery queries that are stored with a cloud peer service at time $T = 700$ secs. Essentially, the queries in the list arrived at a time ≤ 700 and waited for a suitable update query that can meet their provisioning requirements (software, hardware, service type, location). Table 5 depicts an update query that has arrived at $T = 700$. Following the update query arrival, the

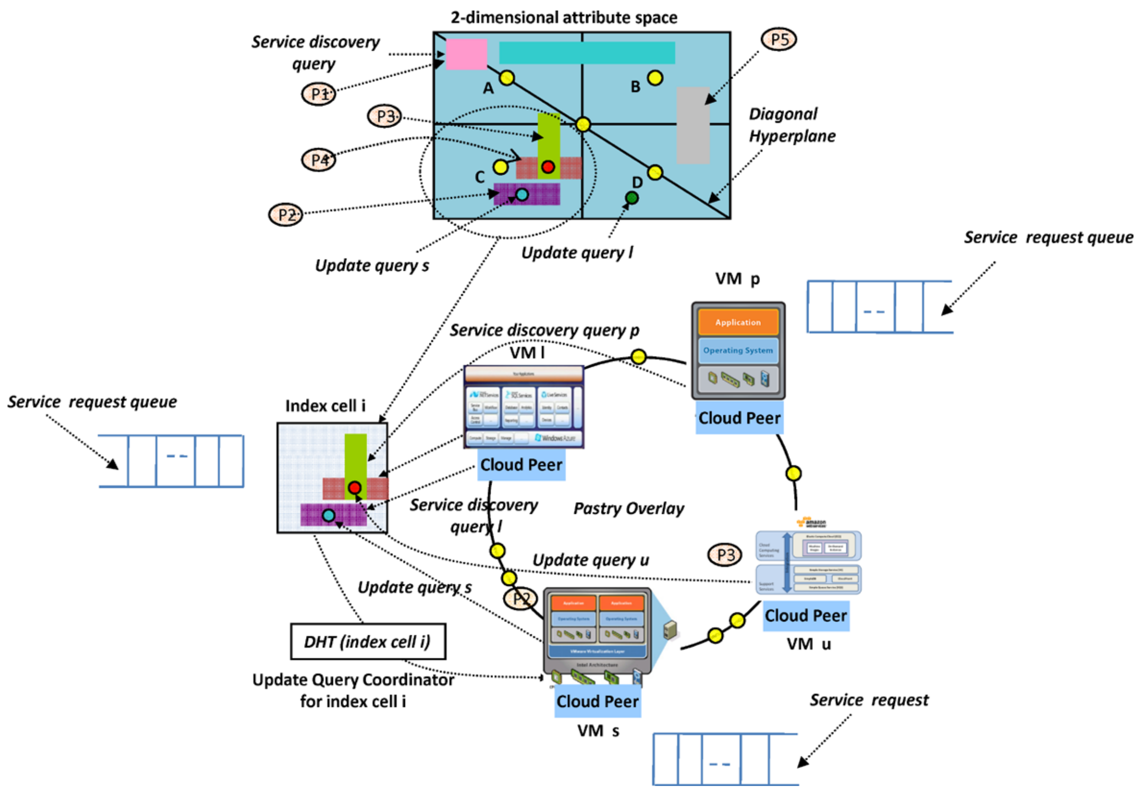


Fig. 5 Peer-to-peer provisioning across service instances. Multi-dimensional service provisioning requests P1, P2, P3, P4, index cell control points A, B, C, D, multi-dimensional update queries l, s and some of the spatial hashings to the Pastry overlay, i.e. the multi-dimensional (spatial) coordinate values of a cell's control point is used as the Pastry key. For this figure $f_{\min} = 2$, $dim = 2$

Table 4 Service discovery query stored with a cloud peer services at time *T*

Time	Query ID	Service type	Speed	Cores	Location
300	Query 1	Web Host-1	> 2	1	USA
400	Query 2	Web Host-2	> 2	1	Singapore
500	Query 3	Web Host-3	> 3	1	Europe

Table 5 A representative Update query published with a cloud Peer service at time *T*

Time	VM IP	Service type	Speed	Cores	Utilization	Location
700	192.168.128.127	Web Host-1	> 2.7	1	70%	USA

cloud peer service undertakes a procedure that allocates the available service capacity with s_i (that published the update query) among the list of matching service discovery queries. Based on the updating s_i's attribute specification, only service discovery query 3 matches. Following this, the cloud peer notifies the provisioning services that posted the Query 1. Note that Queries 2 and 3 have to wait for the arrival of update queries that can match their requirements.

5.3.1 Load-balancing algorithm analysis

This section analyzes the computational tractability of the approach by deriving several time and message complexity bounds to measure the computational quality. Using the example for embarrassingly parallel or Bag of Tasks application models, we analyze the time complexity involved with mapping a service request for a task to a service s_i in the overlay. A service request is encapsulated as a discovery query and is injected to the cloud peer overlay using the Algorithm 1. A concise description of the important steps involved with the load-balancing process is shown in Algorithm 2.

We consider a peer-to-peer overlay of cloud-based n services (SaaS-level) and m independent requests that are required to be mapped to one of these services in a load-balanced manner. This implies that there are total m requests that are submitted by application provisioners (on-behalf of end-users) to the peer-to-peer overlay.

When a discovery query, $d_{i,j}$ arrives at a cloud peer, it is added to the existing query list (see lines 1.17–1.21 in Algorithm 2). When a update query, u_i arrives (refer to lines 1.22–1.32 in Algorithm 2) at a cloud peer, the list of discovery queries that overlap or match (refer to lines 1.6–1.16 in Algorithm 2) with the submitted update query in the d-dimensional space is computed. The overlap signifies that the service requests associated with discovery queries can be served by the update query issuer service, subject to its availability.

In order to compute a load-balanced mapping for requests to services, the cloud peer first, selects the discovery queries stored in the *QueryList* (see line 1.19 in Algorithm 2) in first come first serve order (see line 1.12 in Algorithm 2); then from this list, the number of discovery queries that overlap with a update query is stored in the *QueryList*$_m$ (see line 1.12 of Algorithm 2). The service requests related to the discovery queries are mapped to the service s_i until that service is not over-provisioned

Algorithm 2: Load-balancing undertaken by a cloud peer in the overlay

```
1.1  Input                          : discovery $d_{i,j}$, update $u_i$
1.2  $QueryList \leftarrow \Phi$ ;
1.3  index $\leftarrow 0$ ;
1.4  Output                         : load-balanced mapping
1.5  Definitions                    :
     – Overlap (discovery $d_{i,j}$, update $u_i$): returns true if discovery query $d_{i,j}$
       matches against the update query $u_i$;
     – $QueryList$: stores the list of discovery queries currently mapped to a cloud
       peer $i$;
     – $QueryList_m$: stores the list of discovery queries that matches against a currently
       published update query $u_i$;

     Procedures()                   :
1.6  Match($d_{i,j}$, $u_i$)        :
1.7  Input                          : discovery $d_{i,j}$, update $u_i$
1.8  index $\leftarrow 0$ ;
1.9  $QueryList_m \leftarrow \Phi$ ;
1.10 foreach discovery $d_{i,j} \in QueryList$ do
1.11     if Overlap (discovery $d_{i,j}$, update $u_i$ ) $\neq \Phi$ then
1.12         $QueryList_m \leftarrow QueryList_m$ $[index_m$ $] \cup d_{i,j}$ ;
1.13         $index_m \leftarrow index_m + 1$ ;
1.14     end
1.15 end
1.16 return($QueryList_m$) ;
1.17 Discovery Query Arrival($d_{i,j}$):
1.18 Input                          : discovery $d_{i,j}$
1.19 $QueryList$ [index ] $\leftarrow \cup\, d_{i,j}$ ;
1.20 $index_m \leftarrow index_m + 1$ ;
1.21 return;
1.22 Update Query Arrival($u_i$)    :
1.23 Input                          : update $u_i$ from service $s_i$
1.24 $QueryList_m \leftarrow \Phi$ ;
1.25 $QueryList_m \leftarrow$ Match($d_{i,j}$, $u_i$) ;
1.26 counter $\leftarrow$ SizeOf ($QueryList_m$) ;
1.28 while $s_i$ is not over-provisioned $OR$ counter $\geq 0$ do
1.29     map reqeust $QueryList_m$ [counter ] to $s_i$;
1.30     counter $\leftarrow$ counter + 1 ;
1.31 end
1.32 return;
```

(lines 1.28–1.30 in Algorithm 2). In worst case, *QueryList* in Algorithm 2 can contain m number of discovery queries and all of them could potentially match to a update query, u_i. Hence, in this case, complexity involved with calculating (lines 1.11–1.14) the match list and mapping (lines 1.28–1.31) of requests to services is $O(m)$.

The load-balancing procedure can utilize the dynamic resource parameters such as the number of available processors, queue length, etc. as the over-provision indicator. These over-provision indicators are encapsulated in the update query object by the services. The next section describes the overall message complexity involved with Algorithm 1 and Algorithm 2.

5.4 Message complexity analysis

Lemma 1 *In a overlay of n heterogeneous services, on average a service request $d_{i,j}$ requires $O(\log n)$ messages to be sent in order to locate a service that can successfully complete the request without being overloaded.*

Mapping a service request, which is encapsulated in a discovery query message $d_{i,j}$, involves the following steps: (i) posting a discovery query the cloud peer overlay and (ii) mapping the service request to the matching service (refer to Algorithm 2). Hence, the total number of messages produced in successfully allocating a request to a service is summation of the number of messages produced in these two steps. In rest of this discussion, the terms discovery query and service request are used interchangeably.

We denote the number of messages generated in mapping a service request $d_{i,j}$ by a random variable M_c. The distribution of M_c is function of the problem parameters including query size, dimensionality of search space, query rate, division threshold, and data distribution. Note that the derivation presented in this paper assumes that the Pastry method is used for delegation of service messages in the overlay. Essentially, a control point at the $f_{\min}$ level of the logical d-dimensional Cartesian space can be reached in $O(\log_b n)$ routing hops using the Pastry routing method (see lines 0.11–0.15 in Algorithm 1). Based on the above discussion, in order to compute the worst case message lookup and routing complexity one additional random variable $T_{d_{i,j}}$ is considered. $T_{d_{i,j}}$ denotes the number of disjoint query path undertaken in mapping a discovery query. The d-dimensional index described in Sect. 5.2 maps a service request to utmost mapped to 2 index cells. Note that the number of index cells to which a service request can be mapped is dependent on the spatial index. With Pastry DHT method every disjoint path will undertake $E[T_{d_{i,j}}] \times (\log_b\ n)$ routing hops with high probability. Hence, the expected value of $M_{d_{i,j}}$ is given by:

$$E[T_{d_{i,j}}] = E[M_{d_{i,j}}] \times (\log_b\ n),$$

substituting $E[T_{d_{i,j}}]$ with the value 2 and adding 1 for messages involved with actual mapping,

$$E[M_{d_{i,j}}] = 2 \times (\log_b n) + 1,$$

ignoring the constant terms in the above equation we get,

$$E[M_{d_{i,j}}] = O(\log n). \tag{1}$$

The above equation shows that mapping message complexity function growth rate is bounded by the function $O(\log n)$.

Lemma 2 *In a overlay of n services and total m service requests, the total mapping messages (see lines* 0.11–0.16 *in Algorithm* 1*) generated in the system is bounded by the function* $O(m \times n \times \log n)$.

This lemma directly follows from definition of Algorithm 1. Since a service in the system requires $O(\log n)$ messages to be undertaken before it can be successfully allocated to service, therefore computing the mapping message complexity for m requests is straightforward.

Lemma 3 *In a overlay of n heterogeneous services, each service posts p update query over a time period t, then the average-case message complexity involved with mapping (refer to lines* 0.11–0.16 *in Algorithm* 1*) these update queries to cloud peers in the overlay is bounded by the function* $O(E[T_{u_i}] \times p \times \log n)$.

The proof for this definition directly follows from Lemma 1. The procedure for mapping the update query to cloud peers is similar to the one followed for a discovery query. A update query is always associated with an event region, and all index cells that fall fully and partially within the even region will be selected to receive the corresponding update query. The number of disjoint query path taken to map a update query is denoted by random variable T_{u_i} with mean $E[T_{u_i}]$.

6 Experiments and evaluation

In this section, we evaluated the performance of the proposed peer-to-peer cloud provisioning approach (cloud peer) by creating a overlay of services that are deployed across virtual machines within the Amazon EC2 infrastructure. We assumed unsaturated server availability for these experiments, so that enough capacity could always be allocated to a VM for any service request. Next, we describe the various details related to cloud Peer (peer-to-peer network, multidimensional index structure, and network configuration parameters) setup, PaaS layer provisioning software, and application model related to this study.

6.1 Cloud peer details

A cloud peer service operates at PaaS (refer to Fig. 4) layer and handles activities related to decentralized query (discovery and update) routing, management, and matching. Additionally, it also implements the higher level services such as publish/subscribe based loosely-coupled interactions and service selections. Every VM instance, which is deployed on the Amazon EC2 platform, hosts a number of services and a cloud peer service that loosely glues it to the overlay. Next, follows the details related to the configuration of cloud peer.

FreePastry [5] *network configuration*: Both cloud peers' nodeIds and discovery/update queries' overlay IDs were randomly generated from the 160-bit Pastry identifier space. These nodeIds and overlay IDs were uniformly distributed in the identifier space. Every cloud peer service was configured to buffer maximum of 1,000 messages at a given instance of time. The buffer size was chosen to be sufficiently large such that the FreePastry framework did not drop any messages. Other network parameters were configured tp the default values as given in the file *freepastry.params*. This file is generally provided with the FreePastry distribution.

Multidimensional index configuration: The minimum division $f_{\min}$ of logical multi-dimensional index was set to 3, while the maximum height of the distributed tree (multi-dimensional index), $f_{\max}$ was constrained to 3. In other words, the division of the multi-dimensional attribute space was not allowed beyond $f_{\min}$. This is done for keeping the setup simple and easy to configure. The multidimensional index tree (space) had provision for defining service discovery and update queries that specify the service characteristics in 4 dimensions including application service types, number of processing cores available on the server hosting the VM, hardware architecture of the processor(s), and their processing speed. The aforementioned multidimensional index configuration results into $81(3^4)$ index cells at $f_{\min}$ level.

Service discovery and update queries multi-dimensional extent: Update queries, which are posted by service instances, express equality constraints on its type, installed software environments, and hardware configuration attribute values (e.g., =).

6.2 Aneka: PaaS layer application provisioning and management service

At PaaS layer, we utilize the Aneka [11] software framework that handles activities related to application element scheduling, execution, and management. Aneka is a .NET-based service oriented platform for constructing cloud computing environments. To create a cloud application provisioning systems using Aneka, a developer or application scientist needs to start an instance of the configurable Aneka container hosting required services on each selected VMs.

Services of Aneka can be broadly characterized into two distinct spaces: (i) Application Provisioner: This service implements the functionality that: (i) accepts application workload from cloud users; (ii) performs dynamic discovery of application management services via the *cloud peer service*; (iii) dispatches workload to application management service; (iv) monitors the progress of their execution; and (v) collects the output data, which returned back to the cloud users. An Application Provisioner need not be hosted within a VM; it only needs to know the endpoint address (such as web service address) of a random cloud peer service in the overlay to which it can connect and submit its service discovery query; and (ii) Application Management Service: This service, which is hosted within a VM, is responsible for handling execution and management of submitted application workload. An application management service sits within a VM and *updates its usage status, software, and hardware configurations by sending update queries to the Cloud peer overlay*. One or more instance of application management service can be connected in a single-level hierarchy to be controlled by a root level Application Provisioner. This kind of service integration is aimed at making application programming flexible, efficient, and scalable.

6.3 Test application

The PaaS layer software service, Aneka, supports composition, orchestration, and execution of application programs that are composed using different application programming models to be executed within the same software environment. The experimental evaluation in this paper considers execution of applications programmed using thread model. The thread programming model defines an application as a collection of one or more independent work units. This model can be successfully utilized to compose and program embarrassingly parallel programs (parameter sweep applications). The thread model fits well for implementing and architecting new applications, algorithms on cloud infrastructures since it gives finer degree of control and flexibility as regards to runtime control.

To demonstrate the feasibility of architecting cloud provisioning services based cloud peer overlay, we consider composition and execution of *Mandelbrot Set computation*. Mathematically, the Mandelbrot set is an ordered collection of points in the complex plane, the boundary of which forms a fractal. The Application Provisioner service implements and enables the Mandelbrot fractal calculation on cloud resources, using a the thread programming model. The application submission interface allows the end-users to configure number of horizontal and vertical partitions into which the fractal computation can be divided. These horizontal and vertical partitions, we refer to as *problem complexity*. The number of independent thread units created is equal to the *horizontal* × *vertical* partitions. For evaluations, we vary the values for horizontal and vertical parameters over the intervals, 5 × 5, 10 × 10, 15 × 15, and 20 × 20.

6.4 Deployment of test services on amazon EC2 platform

In order to test the feasibility of aforementioned services in regards to the provisioning of application services on Amazon EC2 cloud platform, we created Amazon Machine Images (AMIs) packaged with a cloud peer, Aneka's Application Provisioner and Management services. The image that hosted the cloud peer and Aneka Provisioner services was equipped with Microsoft Windows Server 2008 R1 SP2 Datacenter edition, Microsoft SQL Server 2008 Express, Internet Information Services 7, Tomcat 6.0.10, and Axis2 1.2 container. On the other hand, while the image that hosted the Aneka Management Service had Microsoft Windows Server 2008 R1 SP2 Datacenter system installed, cloud peer service was exposed to the provisioning and management services through WS* interfaces. Later, we built the customized Amazon Machine Images from the aforementioned basic images. We configured three Application Provisioners and nine Management Services. The Management Service was divided into groups of three that connected to a common Application Provisioner, this resulted in a hierarchical structure. The Application provisioner services communicate and internetwork through the cloud peer overlay. Figure 6 shows the pictorial representation of the test service setup.

6.5 Results and observations

In this study: (i) the problem complexity was varied over the range 5 × 5, 10 × 10, 15 × 15, and 20 × 20; (ii) heartbeat interval was selected from the following set of

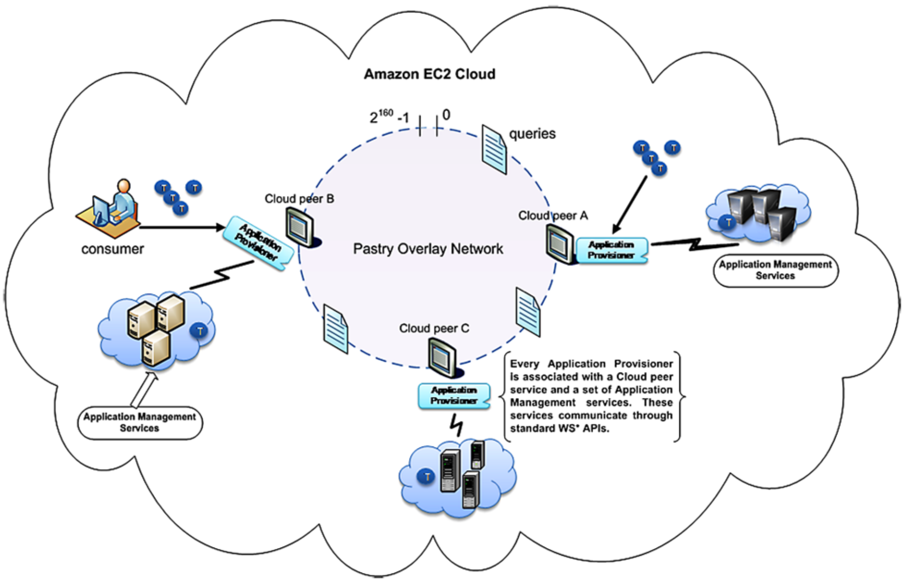

Amazon EC2 Cloud
2^{160} -1
0
queries
Cloud peer B
Cloud peer A
Cloud peer C
Pastry Overlay Network
consumer
Application Provisioner
Application Provisioner
Application Provisioner
Application Management Services
Application Management Services
Every Application Provisioner is associated with a Cloud peer service and a set of Application Management services. These services communicate through standard WS* APIs.
T

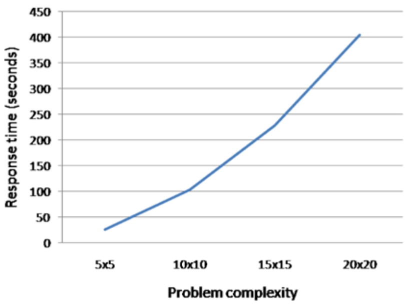
Fig. 6 Test service setup in Amazon EC2

values 1, 10, and 60 seconds; and (iii) the minimum division was selected from the interval [2, 4] in step of 1. The graphs in Figs. 7, 8, and 9 show the performance of the peer-to-peer cloud provisioning techniques (service discovery & load-balancing) in terms of problem complexity, heartbeat interval, and minimum division perspective, respectively.

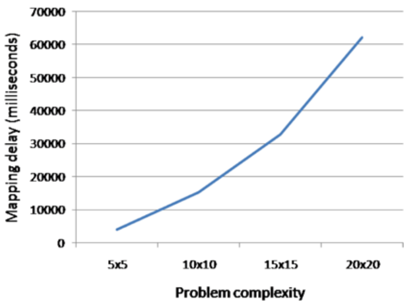
All the application workload were concurrently submitted to the Application Provisioners. Several metrics were quantified for identifying the performance of peer-to-peer cloud provisioning techniques, including response time, query mapping delay, and cloud peer overlay network message complexity for routing multi-dimensional queries. The *response time* for an application was calculated by subtracting the output arrival time of the last thread in the submission list from the time at which the application was originally submitted. On the other hand, *cloud peer overlay message complexity* measures the finer details related number of messages that flow through the network in order to: (i) initialize the multidimensional attribute space, (ii) map the discovery and update queries, (iii) maintain the overlay, and (iv) send notifications. The metric *mapping delay* measures the *average latency* for mapping a service discovery query to a cloud peer.

6.5.1 Problem complexity perspective

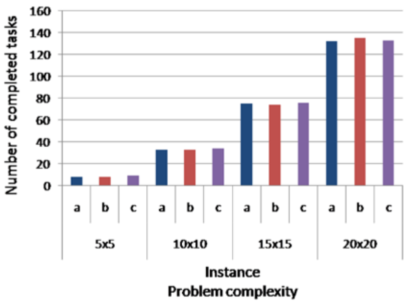
Figure 7 shows the results for response time in seconds, number of tasks and messages, as well as mapping delay with increasing problem complexity size. Cloud consumers submit their applications with a Application Provisioner (see Fig. 6). The initial experimental results in Fig. 7(a) and Fig. 7(b) reveal that with increase in problem complexity (number of *horizontal* × *vertical* partitions of Madelbrot set), the application end-users experience increase in response times and mapping delay. The basic



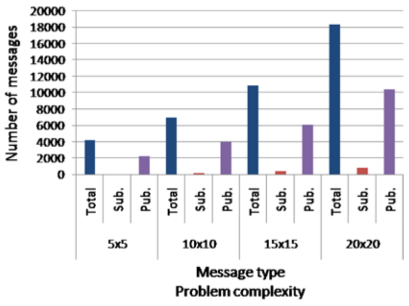
(a) Response time vs. Problem complexity



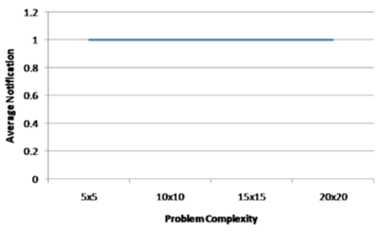
(b) Mapping delay vs. Problem complexity



(c) Number of tasks vs. Problem complexity



(d) Number of messages vs. Problem complexity



(e) Number of average notifications vs. Problem complexity

Fig. 7 The performance of the peer-to-peer cloud provisioning techniques with varying problem complexity. The heartbeat interval was set to 2 seconds and $f_{\min}$ and $f_{\max}$ to 3, for these set of experiments

reason behind this behavior of the cloud system was the fixed number Application Management services (static infrastructure capacity), i.e., with increase in the problem complexity, the number of task threads (a task thread represents a single work unit, e.g., for 5×5 Mandelbrot configuration resulted in creation of 25 task threads) that are required to be serviced with Management services increase, hence leading to worsening queuing and waiting delays. However, this behavior of the system can be fixed through implementation of reactive provisioning of new service instances to counter the effect of increase in problem complexity or size.

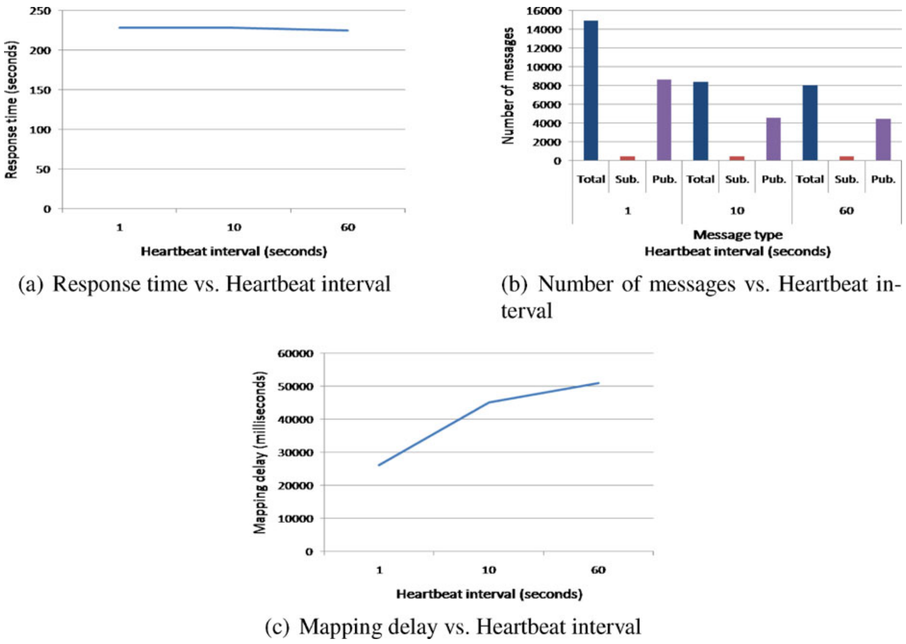
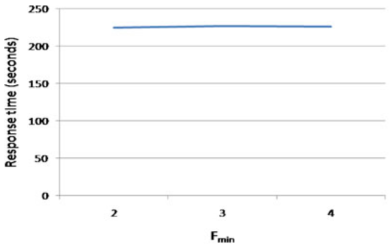


Fig. 8 The performance of the peer-to-peer cloud provisioning technique in terms of heartbeat interval perspective with problem complexity size as 15 × 15 and $f_{\min}$ and $f_{\max}$ as 3

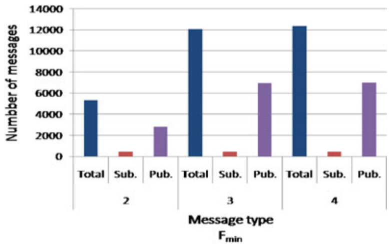
Figure 7(b) presents the measurements for average mapping delay for each discovery query with respect to increase in the problem complexity. The results show that at higher problem complexity, the discovery queries experience increased mapping delay. This happens due to the reason that the discovery queries have to wait for longer period of time before they are matched against an update query (service contention problem). However, the task thread processing time (CPU time) is not affected by the mapping delay, hence the response time in Fig. 7(a) shows no or little change for different problem sizes.

Figure 7(c) indicates the total number completed tasks at each Aneka's Application Management service instance with the increase in problem complexity. For every group of problem complexity, the number of tasks at each instance showed a similar trend. The result shows that the task processing load was evenly distributed across the management service instances. This behavior basically proved that the load-balancing algorithm undertaken by cloud peer services was extremely effective distributing the workload across services instances.

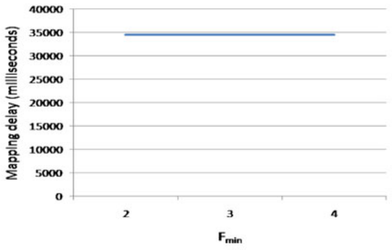
Figure 7(d) shows the total message overhead involved with management of multidimensional index, routing of discovery(marked as sub Fig. 7(d)) and update query (marked as pub) messages, and maintenance of Pastry overlay. We can clearly see that as problem complexity increases the number of messages required for mapping the queries to the overlay network increase as well. The number of discovery and update messages produced in the overlay is a function of the multidimensional index structure that maintains and routes these queries in a peer-to-peer fashion. In addition, from Fig. 7(d), it is evident that our system is scalable since the total number



(a) Response time vs. f_{min}



(b) Number of messages vs. f_{min}



(c) Mapping delay vs. f_{min}

Fig. 9 The performance of the peer-to-peer cloud provisioning technique in terms of maximum division perspective with problem complexity size as 15×15, heartbeat interval as 2 seconds and $f_{\max}$ as 5

of messages is increased linearly with respect to problem size. Hence, the choice of the multidimensional data indexing structure and routing technique governs the manageability and efficiency of the overlay network (latency and messaging overhead). Hence, there is much work required in this domain as regards to evaluating the performance of different types of multidimensional indexing structures for mapping the query messages in peer-to-peer settings.

The proposed load-balancing approach was also highly successful in reducing the number of mapping iterations undertaken for the successful submission of a task to a Application Management service (see Fig. 7(e)). Note that, our results showed that the negotiation and notification complexity of the proposed approach had similar bounds as that of a centrally provisioned system, such as the Amazon EC2 load-balancer or Microsoft Fabric Controller. Essentially, with a centrally provisioned system, the mapping iteration and notification message complexity is $O(1)$. Experiment results (refer to Fig. 7(e)) showed that Aneka's Application Provisioner service in the system receive an average 1 notifications on per task basis. This suggests that the mapping iteration and notification complexity involved with our scheduling technique is $O(1)$.

6.5.2 Heartbeat interval perspective

Figure 8 consists of results for group of test that measured the effect of varying heartbeat (update query publish delay) interval on response time, number of messages and mapping delay. The heartbeat interval was varied as 1 second, 10 seconds, and 60 seconds. At the same time, f_{min} and f_{max} were set to 3 and the problem complexity size was configured to be 15×15. The primary results in Fig. 8(a) showed a near constant response time behavior, which was around 228 seconds, over three heartbeat intervals.

Figure 8(b) depicts the number of messages generated with respect to increasing heartbeat interval. We can see that as heartbeat interval increases, the number of messages generated during the experiment period decrease. For instance, when heartbeat interval was configured to 1 second, the total number of messages produced was 14,974. However, the number of messages dropped significantly for higher values of heartbeat interval. Figure 8(c) reveals the relationship of the mapping delay and the heartbeat interval. We can see that with increase in heartbeat interval, the mapping delay increases to some extent. Therefore, the update query interarrival delay (heartbeat interval) should be chosen in such a way that a balance between mapping delay and message overhead can exist in the system.

6.5.3 Minimum division (f_{min}) perspective

In this experiment (Fig. 9), we were interested in studying the behavior of response time, number of messages and mapping delay parameters with increasing f_{min} value for multi-dimensional index. We can recall that, f_{min} affects the number of index cells that created, which subsequently controls the number of messages required for mapping queries. For this experiment, f_{max} was fixed to 5, while the f_{min} was varied over the interval $[2, 4]$ in step of 1. Next, the heartbeat interval was fixed to 2 seconds. The Mandelbrot application problem complexity was set to 15×15. Our goal in this experiment was to study the effect of f_{min} on the application response time and query mapping delay.

As shown in Fig. 9(a), with the increase in f_{min}, we did not observe a significant change in the average response time. In fact, the response time kept steady when f_{min} varies. At $f_{\text{min}} = 2$, 3 and 4, the response time remains at the value of 226 ± 1.

We also analyzed the number of messages that were required to: (i) manage multi-dimensional index; (ii) route discovery and update query messages; and (iii) maintain Pastry overlay with changing f_{min}. The results for this test is shown in Fig. 9(b). The number of discovery messages (shown as sub in the Fig. 9(b)) remained constant at 450. This was because, number of message required for mapping a discovery is independent of f_{min} value, as discussed in previous sections. Meanwhile, the number of update query mapping messages were found to be doubling with increase in f_{min}. We also observed that changing f_{min} value did not affect average mapping delays (refer to Fig. 9(b)) of discovery queries. At $f_{\text{min}} = 2$, the discovery query in the system on average experienced a mapping delay of 34,494.65 milliseconds. While at $f_{\text{min}} = 4$ this value increased to 35,000 milliseconds.

The key lesson here is that, f_{min} value should be chosen in such a way that query processing load is evenly distributed across cloud peers, while having little or not impact on the application performance. Evaluating this aspect of system is subject of our future work.

7 Conclusion

Developing provisioning techniques that integrate application services in a peer-to-peer fashion is critical to exploiting the potential of cloud computing platforms. Architecting provisioning techniques based on peer-to-peer network models (such as DHTs) is significant; Since peer-to-peer networks are highly scalable, can gracefully adapt to the dynamic system expansion (join) or contraction (leave, failure), are not susceptible to single point of failure. To this end, we presented a software fabric called cloud peer that creates an overlay network of VMs and application services for supporting scalable and self-managing service discovery and load-balancing. The functionality exposed by the cloud peer service is very powerful and our experimental results conducted on Amazon EC2 platform confirms that it is possible to engineer and design peer-to-peer cloud provisioning systems and techniques.

As part of our future work, we would explore other multidimensional data indexing and routing techniques that can achieve close to logarithmic bounds on messages and routing state, balance query (discovery, load-balancing, coordination) processing load, preserves data locality, and minimize the metadata. Another important algorithmic and programming challenge in building robust cloud peer services is to guarantee consistent routing, lookup, and information consistency under concurrent leave, failure and join operations by application services. To address these issues, we will investigate robust fault-tolerance strategies based on distributed replication of attribute/query subspaces to achieve a high level of robustness and performance guarantees.

Acknowledgements Dr. Rajiv Ranjan would like to thank the University of New South Wales for funding his position though the strategic e-Research initiative.

References

1. Amazon auto scaling service (2010) http://aws.amazon.com/autoscaling/. Accessed: May 25, 2010
2. Amazon cloudwatch service (2010) http://aws.amazon.com/cloudwatch/. Accessed: May 25, 2010
3. Amazon elastic mapreduce service (2010) http://aws.amazon.com/elasticmapreduce/. Accessed: May 25, 2010
4. Amazon load balancer service (2010) http://aws.amazon.com/elasticloadbalancing/. Accessed: May 25, 2010
5. An open source pastry dht implementation (2010) http://freepastry.rice.edu/FreePastry. Accessed: May 25, 2010
6. Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G, Patterson DA, Rabkin A, Stoica I, Zaharia M (2009) Above the clouds: a Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb
7. Bakhtiari S, Safavi-naini R, Pieprzyk J, Centre Computer (1995) Cryptographic hash functions: a survey. Technical report
8. Balakrishnan H, Frans Kaashoek M, Karger D, Morris R, Stoica I (2003) Looking up data in p2p systems. Commun ACM 46(2):43–48
9. Bharambe AR, Agrawal M, Seshan S (2004) Mercury: supporting scalable multi-attribute range queries. Comput Commun Rev 34(4):353–366
10. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging it platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comput Syst 25(6):599–616

11. Chu X, Nadiminti K, Jin C, Venugopal S, Buyya R (2007) Aneka: next-generation enterprise grid platform for e-science and e-business applications. In: E-SCIENCE '07: proceedings of the third IEEE international conference on e-science and grid computing, Washington, DC, USA. IEEE Computer Society Press, Los Alamitos, pp 151–159
12. Eucalyptus systems (2010) http://www.eucalyptus.com/. Accessed: May 25, 2010
13. Force.com cloud solutions (saas) (2010) http://www.salesforce.com/platform/. Accessed: May 25, 2010
14. Ganesan P, Yang B, Garcia-Molina H (2004) One torus to rule them all: multi-dimensional queries in p2p systems. In: WebDB '04: proceedings of the 7th international workshop on the web and databases, New York, NY, USA. ACM Press, New York, pp 19–24
15. Gillett FE, Brown EG, Staten J, Lee C (2008) Future view: the new tech ecosystems of cloud, cloud services, and cloud computing. Technical report, Forrester Research, Inc
16. GoGrid Cloud Hosting (2010) (f5) load balancer. GoGrid wiki, http://wiki.gogrid.com/wiki/index.php/(F5)-Load-Balancer. Accessed: May 25, 2010
17. Google app engine (2010) http://code.google.com/appengine/. Accessed: May 25, 2010
18. Gupta I, Birman K, Linga P, Demers Al, van Renesse R (2003) Kelips: building an efficient and stable p2p dht through increased memory and background overhead. In: Proceedings of the 2nd international workshop on peer-to-peer systems (IPTPS '03)
19. Gupta A, Sahin OD, Agrawal D, El Abbadi A (2004) Meghdoot content-based publish/subscribe over p2p networks. In: Middleware '04: proceedings of the 5th ACM/IFIP/USENIX international conference on middleware, New York, NY, USA. Springer, New York, pp 254–273
20. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D (1997) Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: STOC '97: proceedings of the twenty-ninth annual ACM symposium on theory of computing, New York, NY, USA. ACM Press, New York, pp 654–663
21. Lee YC, Zomaya A (2010) Rescheduling for reliable job completion with the support of clouds. Future Gener Comput Syst. doi:10.1016/j.future.2010.02.010
22. Li J, Stribling J, Gil TM, Morris R, Frans Kaashoek M (2005) Comparing the performance of distributed hash tables under churn. In: Lecture notes in computer science. Springer, Berlin, pp 87–99
23. Lua EK, Crowcroft J, Pias M, Sharma R, Lim S (2005) A survey and comparison of peer-to-peer overlay network schemes. IEEE Commun Surv Tutor 7(2):72–93
24. Mosso cloud platform (2010) http://www.rackspacecloud.com. Accessed: May 25, 2010
25. Openvpn (2010) http://openvpn.net/. Accessed: May 25, 2010
26. Parashar M, Gnanasambandam N, Quiroz A, Kim H, Sharma N (2009) Towards autonomic workload provisioning for enterprise grids and clouds. In: Proceedings of the 10 th IEEE/ACM international conference on grid computing (Grid 2009), pp 50–57
27. Preneel B (1999) The state of cryptographic hash functions. In: Lectures on data security: modern cryptology in theory and practice, 1994. Lecture notes in computer science, vol 1561. Springer, Berlin, pp 158–182
28. Ranjan R (2007) Coordinated resource provisioning in federated grids. PhD thesis, The University of Melbourne
29. Ranjan R, Harwood A, Buyya R (2008) Peer-to-peer-based resource discovery in global grids: a tutorial. IEEE Commun Surv Tutor 10(2):6–33
30. Ratnasamy S, Francis P, Handley M, Karp R, Shenker, S (2001) A scalable content-addressable network. In: SIGCOMM '01: proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, New York, NY, USA. ACM Press, New York, pp 161–172
31. Rejila cloud platform (2010) http://www.rejila.com/. Accessed: May 25, 2010
32. Rochwerger B, Breitgand D, Levy E, Galis A, Nagin K, Llorente IM, Montero R, Wolfsthal Y, Elmroth E, Caceres J, Ben-Yehuda M, Emmerich W, Galan F (2009) The reservoir model and architecture for open federated cloud computing. IBM J Res Dev 53(4). http://dl.acm.org/citation.cfm?id=1850659.1850663
33. Rowstron AIT, Druschel P (2001) Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware '01: proceedings of the IFIP/ACM international conference on distributed systems platforms heidelberg, London, UK. Springer, Berlin, pp 329–350
34. Spence D, Crowcroft J, Hand S, Harris T (2005) Location based placement of whole distributed systems. In: CoNEXT '05: proceedings of the 2005 ACM conference on emerging network experiment and technology, New York, NY, USA. ACM Press, New York, pp 124–134

35. Stoica I, Morris R, Liben-Nowell D, Karger DR, Frans Kaashoek M, Dabek F, Balakrishnan H (2003) Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans Netw 11(1):17–32
36. Tanin E, Harwood A, Samet H (2007) Using a distributed quadtree index in peer-to-peer networks. VLDB J 16(2):165–178
37. The S. Reservoir (2008) Reservoir—an ict infrastructure for reliable and effective delivery of services as utilities. Technical report, IBM Research
38. Varia J (2009) Cloud architectures. Technical report, Amazon Web Services
39. Vpn-cubed (2010) http://www.cohesiveft.com/vpncubed/. Accessed: May 25, 2010
40. Windows azure platform (2010) http://www.microsoft.com/azure/. Accessed: May 25, 2010
41. Zhang X, Freschl JL, Schopf JM (2003) A performance study of monitoring and information services for distributed systems. In: High performance distributed computing. Proceedings 12th IEEE international symposium on, 22–24 2003, pp 270–281