Chapter 12
Peer-to-Peer Cloud Provisioning: Service Discovery and Load-Balancing

Rajiv Ranjan, Liang Zhao, Xiaomin Wu, Anna Liu, Andres Quiroz, and Manish Parashar

Abstract Clouds have evolved as the next-generation platform that facilitates creation of wide-area on-demand renting of computing or storage services for hosting application services that experience highly variable workloads and requires high availability and performance. Interconnecting Cloud computing system components (servers, virtual machines (VMs), application services) through peer-to-peer routing and information dissemination structure are essential to avoid the problems of provisioning efficiency bottleneck and single point of failure that are predominantly associated with traditional centralized or hierarchical approaches. These limitations can be overcome by connecting Cloud system components using a structured peer-to-peer network model (such as distributed hash tables (DHTs)). DHTs offer deterministic information/query routing and discovery with close to logarithmic bounds as regards network message complexity. By maintaining a small routing state of O (log n) per VM, a DHT structure can guarantee deterministic look-ups in a completely decentralized and distributed manner.

This chapter presents: (i) a layered peer-to-peer Cloud provisioning architecture; (ii) a summary of the current state-of-the-art in Cloud provisioning with particular emphasis on service discovery and load-balancing; (iii) a classification of the existing peer-to-peer network management model with focus on extending the DHTs for indexing and managing complex provisioning information; and (iv) the design and implementation of novel, extensible software fabric (*Cloud peer*) that combines public/private clouds, overlay networking, and structured peer-to-peer indexing techniques for supporting scalable and self-managing service discovery and load-balancing in Cloud computing environments. Finally, an experimental evaluation is presented that demonstrates the feasibility of building next-generation Cloud provisioning systems based on peer-to-peer network management and information

R. Ranjan (✉)
SA Project, CRC Smart Services, Service Oriented Computing Research Group,
School of Computer Science and Engineering, University of New South Wales, Australia
e-mail: rajivr@cse.unsw.edu.au

N. Antonopoulos and L. Gillam (eds.), *Cloud Computing: Principles, Systems and Applications*, Computer Communications and Networks,
DOI 10.1007/978-1-84996-241-4_12, © Springer-Verlag London Limited 2010

dissemination models. The experimental test-bed has been deployed on a public cloud computing platform, Amazon EC2, which demonstrates the effectiveness of the proposed peer-to-peer Cloud provisioning software fabric.

12.1 Introduction

Cloud computing [1–3] has emerged as the next-generation platform for hosting business and scientific applications. It offers infrastructure, platform, and software as services that are made available as on-demand and subscription-based services in a pay-as-you-go model to users. These services are, respectively, referred to as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Adoption of Cloud computing platforms [4–9] as an application 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 Cloud services as an effective means for cost cutting by leasing their IT service hosting and maintenance from external cloud providers; (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 publically accessible clouds (such as Amazon EC2 [8]) that expose their capabilities as a network of virtualized services (hardware, storage, database) is known as Cloud provisioning. The Cloud provisioning process consists of two key steps [10]: (i) VM provisioning, involving instantiation of one or more VMs on physical servers hosted within public or private Cloud computing environments – the selection of a physical server for hosting VMs in a cloud is based on a number of mapping requirements including available memory, storage space, and proximity of the parent cloud; and (ii) application service provisioning, with mapping and scheduling of requests to the services that are hosted within a VM or on a set of VMs. In this chapter, we mainly focus on the second step, which involves dynamically distributing the incoming requests among the services in a load-balanced and decentralized manner, given a set of VMs that are hosting different types of application services.

Cloud provisioning from a business services point of view involves deriving cloud-based application component deployments driven by expected performance (Quality of Service (QoS)). Clouds offer an unprecedented pool of software and hardware resources, which gives businesses a unique ability to handle the temporal variation in their service demands through dynamic provisioning or deprovisioning

of capabilities. Whenever there is a variation in temporal and spatial locality of workload such as number of concurrent users, total users, and load conditions, each application component must dynamically scale (application service elasticity) to offer good quality of experience to users, and maintain an optimal usage of cloud resources. Cloud-enabling any class of application service would require developing models for service placement, computation, communication, and storage, with emphasis on important scalability requirements.

Currently, one of the prominent Cloud service providers Amazon EC2 offers two services, namely CloudWatch [11] and Elastic Load Balancer [12]. Fundamentally, CloudWatch and Elastic Load Balancer are centralized web services that can be associated with numerous EC2 instances. However, centralized approaches have several critical design limitations including: (i) single point of failure; (ii) lack of scalability; (iii) high network communication cost at links leading to the service; (iv) requirement of high computational power to serve a large number of resource look-up and updated queries on the server running the central service.

As Clouds become ready for mainstream acceptance, scalability [13] of services will come under more severe scrutiny due to the increasing number of online services in the Cloud, and massive numbers of global users. To overcome the aforementioned limitations, fundamental Cloud services for discovery, monitoring, and load-balancing should be decentralized by nature and different service components (VM instances and application elements) must interact to adaptively maintain and achieve the desired system wide connectivity and behaviour.

The rest of this chapter is organized as follows: First, a layered approach to architecting peer-to-peer Cloud provisioning system is presented. This is followed by some survey results on Cloud provisioning capabilities in leading commercial public clouds. 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 discussion of the design and implementation of peer-to-peer Cloud provisioning (Cloud peer) software fabric. Lastly, we present the analysis and experimental results of the peer-to-peer Cloud provisioning implementation across a public Cloud (Amazon EC2) environment (Table 12.1).

Table 12.1 Summary of provisioning capabilities exposed by public Cloud platforms

Cloud platforms	Load balancing	Provisioning	Autoscaling
Amazon Elastic Compute Cloud	√	√	√
Eucalyptus	√	√	×
Microsoft Windows Azure	√	√ (Fixed templates so far)	√ (Manually at the moment)
Google App Engine	√	√	√
GoGrid Cloud Hosting	√	√	√ (Programmatic way only)

12.2 Layered Peer-to-Peer Cloud Provisioning Architecture

This section presents information on various architectural elements that form the basis for peer-to-peer Cloud provisioning architecture. It also presents an overview of the applications that would benefit from the architecture, which envisages a hosting infrastructure consisting of multiple geographically distributed private and public clouds owned by one or more service providers. Figure 12.1 shows the layered design of the peer-to-peer Cloud provisioning architecture. Physical Cloud servers, along with core middleware capabilities, form the basis for delivering IaaS. The user-level middleware aims at providing PaaS capabilities. The top layer focuses on application services (SaaS) by making use of services provided by the lower layers. PaaS/SaaS services are often developed and provided by third-party service providers, who are 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). These applications have radically different application characteristics and workload profiles, and hence, to cope with the variation in temporal and spatial locality of service request, the application services must be supported by a Cloud provisioning infrastructure that dynamically scales the deployed

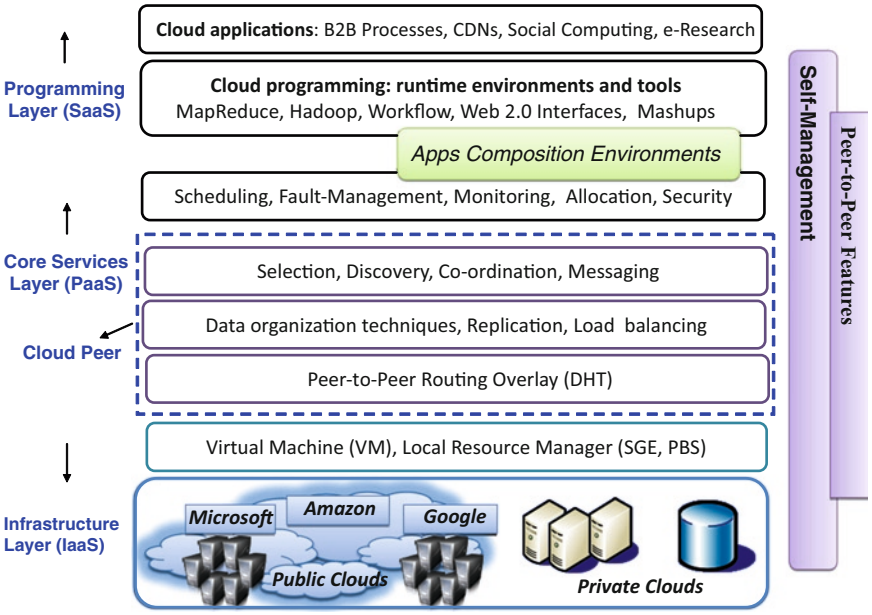

Fig. 12.1 A layered peer-to-peer Cloud provisioning architecture

services in order to achieve good performance, optimal resource usage, and hence offer quality experience to its end-users.

Development Framework Layer: This layer includes the software 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 run-time environment-enabling Cloud computing capabilities to application services built using User-Level Middleware. 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 co-ordinated interaction, 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's fabric controller, and Aneka [14].

To be able to provide support for decentralized service discovery [15] and load-balancing between cloud components (VM instances, application services), novel distributed hash table (DHT)-based PaaS layer services, techniques, and algorithms need to be developed at this layer for supporting complex interactions with guarantees on dynamic management. In Fig. 12.1, this component of PaaS layer is shown as Cloud peer service. Architecting Cloud services based on decentralized network models or overlays (such as DHTs) is significant since DHTs are highly scalable, can gracefully adapt to the dynamic system expansion (new host/VM/service instantiation) or contraction (host/VM/service instance destruction) and outage, and are not susceptible to single point of failure in massive scale, internetworked private and public cloud environments.

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 exist 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 behaviour and the isolation of security contexts.

Another trend in Cloud usage is combination of private clouds with public clouds, in order to attend unexpected or periodic peaks in local demand without investing in acquiring new equipment for the local infrastructure. Resources from the data center may be either available for public in general (public clouds) or may be restricted to users belonging to the organization that owns the data center (private clouds). It is also possible to have hybrid models, in which resources are leased from the public cloud whenever the private cloud cannot cope with the incoming demand.

12.3 Current State-of-the-Art and Practice in Cloud Provisioning

Key players in public Cloud computing, including Amazon, Microsoft, Google App Engine, Eucalyptus [16], and GoGrid, offer a variety of prepackaged services for monitoring, managing, and provisioning resources. However, the techniques implemented in each of these Clouds vary.

The three Amazon Web Services (AWS), Elastic Load Balancer [12], Auto Scaling [17], and CloudWatch [11], together expose functionalities that are required for undertaking provisioning of application services on Amazon EC2. Elastic Load Balancer service automatically provisions incoming application workload across available Amazon EC2 instances. Auto-scaling service can be used to dynamically scale-in or scale-out the number of Amazon EC2 instances for handling changes in service demand patterns. And finally, the CloudWatch service can be integrated with the above services for strategic decision-making based on collected real-time information.

Eucalyptus is an open source Cloud computing platform. It is composed of three controllers. Among the controllers, the cluster controller is a key component to application service provisioning and load balancing. Each cluster controller is hosted on the head node of a cluster to interconnect outer public networks and inner private networks together. By monitoring the state information of instances in the pool of server controllers, the cluster controller can select the available service/server for provisioning incoming requests. However, when compared with AWS, Eucalyptus still lacks some of the critical functionalities, such as autoscaling for built-in provisioner.

Fundamentally, Windows Azure Fabric has a weave-like structure, which is composed of nodes (servers and load balancers), and edges (power, Ethernet, and serial communications). The fabric controller manages a service node through a built-in service, the Azure fabric controller agent, which runs in the background tracking the state of the server and reporting these metrics to the controller. If a fault state is reported, the controller can manage a reboot of the server or a migration of services from the current server to other healthy servers. Moreover, the controller also supports service provisioning by matching the services/VMs that meet the required demands.

GoGrid Cloud Hosting offers developers F5 Load Balancers [18] for distributing application service traffic across servers, as long as IPs and specific ports of these servers are attached. The load balancer provides the Round Robin algorithm and Least Connect algorithm for routing application service requests. Also, the load balancer is able to sense a crash of the server, redirecting further requests to other available servers. But currently, GoGrid Cloud Hosting only gives developers programmatic APIs to implement their custom autoscaling service.

Unlike other Cloud platforms, Google App Engine offers developers a scalable platform in which applications can run, rather than providing access directly to a customized virtual machine. Therefore, access to the underlying operating system

is restricted in App Engine. Load-balancing strategies, service provisioning, and autoscaling are all automatically managed by the system behind the scenes.

In addition, no single Cloud infrastructure provider has its data centers at all possible locations throughout the world. As a result, Cloud application service (SaaS) providers will have difficulty in meeting QoS expectations for all their users. Hence, they would like to logically construct hybrid Cloud infrastructures (mixing multiple public and private clouds) to provide better support for their specific user needs. This kind of requirement often arises in enterprises with global operations and applications such as Internet service, media hosting, and Web 2.0 applications. This necessitates building technologies and algorithms for seamless integration of Cloud infrastructure service providers for provisioning of services across different Cloud providers.

12.4 Cloud Service Discovery and Load-Balancing Using DHT Overlay

12.4.1 Distributed Hash Tables

Structured systems such as DHTs offer deterministic query search results within logarithmic bounds on network message complexity. Peers in DHTs such as Chord, CAN, Pastry, and Tapestry maintain an index for O (log n) peers where n is the total number of peers in the system. Inherent to the design of a DHT are the following issues [19]: (i) generation of node-ids and object-ids, called keys, using cryptographic/randomizing hash functions such as SHA-1 [19–22] – the objects and nodes are mapped on the overlay network depending on their key value and each node is assigned responsibility for managing a small number of objects; (ii) building up routing information (routing tables) at various nodes in the network – each node maintains the network location information of a few other nodes in the network; and (iii) an efficient look-up query resolution scheme.

Whenever a node in the overlay receives a look-up request, it must be able to resolve it within acceptable bounds such as in O (log n) routing hops. This is achieved by routing the look-up request to the nodes in the network that are most likely to store the information about the desired object. Such probable nodes are identified by using the routing table entries. Though at the core various DHTs (Chord, CAN, Pastry, and Tapestry, etc.) are similar, still there exist substantial differences in the actual implementation of algorithms including the overlay network construction (network graph structure), routing table maintenance, and node join/leave handling. The performance metrics for evaluating a DHT include fault-tolerance, load-balancing, efficiency of look-ups and inserts, and proximity awareness [23]. In Table 12.2, we present the comparative analysis of Chord, Pastry, CAN, and Tapestry based on basic performance and organization parameters. Comprehensive details about the performance of some common DHTs under churn can be found in [24].

Table 12.2 Summary of complexity of distributed hash table overlays

DHT system	Overlay structure	Look-up protocol	Network parameters	Routing table size	Routing complexity	Join/ leave overhead
Chord	Circular identifier space	Matching key and server-id	n =number of servers	$O(\log n)$	$O(\log n)$	$O((\log n)^2)$
Pastry	Plaxton style mesh	Matching key and prefix in server-id	n =number of servers in the network, b= base of the identifier	$O(\log_b n)$	$O(b \log_b n) + b$	$O(\log n)$
CAN	Multidimensional space	Key, value pair map to a point in space	n =number of servers in the network, d = dimensions	$O(2\,d)$	$O(d\,n^{1/d})$	$O(2\,d)$
Tapestry	Plaxton style mesh	Matching suffix in server-id	n =number of servers in the network, b = base of the identifier	$O(\log_b n)$	$O(b \log_b n) + b$	$O(\log n)$

Other classes of structured peer-to-peer systems such as Mercury [25] do not apply randomizing hash functions for organizing data items and nodes. The Mercury system organizes nodes into a circular overlay and places data contiguously on this ring. As Mercury does not apply hash functions, data partitioning among nodes is not uniform. Hence, it requires an explicit load-balancing scheme. In recent developments, new-generation P2P systems have evolved to combine both unstructured and structured P2P networks. We refer to this class of systems as hybrid. Structella [26] is one such P2P system that replaces the random graph model of an unstructured overlay (Gnutella) with a structured overlay, while still adopting the search and content placement mechanism of unstructured overlays to support complex queries. Other hybrid P2P design includes Kelips [27] and its variants. Nodes in Kelips overlay periodically gossip to discover new members of the network, and during this process nodes may also learn about other nodes as a result of look-up communication. Other variants of Kelips allow routing table entries to store information for every other node in the system. However, this approach is based on the assumption that the system experiences low churn rate [24]. Gossiping and one-hop routing approach has been used for maintaining the routing overlay in the work [28].

12.4.2 Designing Complex Services over DHTs

Limitations of Basic DHT Implementations and Query Types: Traditionally, DHTs have been efficient for single-dimensional queries such as "finding all resources that match the given attribute value." Since Cloud computing IaaS and PaaS level services such as servers, VMs, enterprise computers (private cloud resources), storage devices, and databases are identified by more than one attribute, a search query for these services is always multidimensional. These search dimensions or attributes can include service type, processor speed, architecture, installed operating system, available memory, and network bandwidth.

Based on recent information published by Amazon EC2 CloudWatch service, each Amazon Machine Image (AMI) instance has seven performance metrics (see Table 12.3) and four dimensions (see Table 12.4) associated with it. Additionally, these AMIs can host different application service types, including web hosting,

Table 12.3 Performance metrics associated with an Amazon EC2 AMI instance

CPU Utilization	Network Incoming Traffic	Network Outgoing Traffic	Disk Write Operations	Disk Read Operations	Disk Write Bytes	Disk Read Bytes

Table 12.4 Performance dimensions associated with an Amazon EC2 AMI instance

Image ID	Autoscaling group name	Instance ID	Instance type

social networking, content-delivery, and high-performance computing, that have varying request invocation, access, and distribution patterns. The type of application services hosted by an AMI instance is dependent on the business needs and scientific experiments. In these cases, a Cloud service discovery query (which can be issued by provisioning software) will combine the aforementioned attributes related to AMI instances and application service types and therefore can have the following semantics:

> Cloud Service Type = "web hosting" && Host CPU Utilization < "50%" && Instance OSType = "WinSrv2003" && Host Processor Cores > "1" && Host Processors Speed > "1.5 GHz" && Host Cloud Location = "Europe"

On the other hand, VM instances deployed on the Cloud hosts needs to publish their information so that provisioning software can search and discover them. VM instances update their software and hardware configuration and the deployed services' availability status by sending *update query* to the DHT overlay. An 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"

Extending DHTs to support indexing and matching of multidimensional range (service discovery query) or point (update query) queries, to index all resources whose attribute value overlaps a given search space, is a complex problem. Multidimensional range queries are based on ranges of values for attributes rather than on specific values. Compared to single-dimensional queries, resolving multidimensional queries is far more complicated, as there is no obvious total ordering of the points in the attribute space. Further, the query interval has varying size, aspect ratio, and position such as a window query. The main challenges involved in enabling multidimensional queries in a DHT overlay include designing efficient service attribute data: (i) distribution or indexing techniques; and (ii) query routing techniques.

Data Indexing Techniques for Mapping Multidimensional Range and Point Queries: A data indexing technique partitions the multidimensional attribute space over the set of VMs in a DHT network. Efficiency of the distribution mechanism directly governs how the query processing load is distributed among the Cloud peers. A good distribution mechanism should possess the following characteristics [29]: (i) locality: data points nearby in the attribute space should be mapped to the same Cloud peer, hence limiting the distributed lookup complexity; (ii) load balance: the number of data points indexed by each Cloud peer should be approximately the same to ensure uniform distribution of query processing; (iii) minimal metadata: prior information required for mapping the attribute space to the overlay space should be minimal; and (iv) minimal management overhead: during VM instantiation and destruction operation, update policies such as the transfer of data points to a newly joined Cloud peer should cause minimal network traffic. Note that the assumption here is that every VM instance hosts a Cloud peer service, which is responsible for managing activities related to overlay network.

There are different kinds of database indices [30] that can handle mapping of multidimensional objects such as the space filling curves (SFCs) (including the Hilbert curves, Z-curves), k-d tree, MX-CIF Quad tree, and R*-tree in a DHT overlay. In literature, these indices are referred to as spatial indices [31]. Spatial indices are well suited for handling the complexity of multidimensional queries. Although some spatial indices can have issues as regards to routing load-balance in case of a skewed attribute/data set, all the spatial indices are generally scalable in terms of the number of hops traversed and messages generated while searching and routing multidimensional/spatial service discovery and update queries. However, there are different tradeoffs involved with each of the spatial indices, but basically they can all support scalability and Cloud service discovery. Some spatial index would perform optimally in one scenario but the performance could degrade if the attribute/data distribution changed significantly.

Routing Techniques for Handling Multidimensional Queries in DHT Overlay: DHTs guarantee deterministic query look-up with logarithmic bounds on network message cost for single-dimensional queries. However, Cloud's service discovery and update query are multidimensional (as discussed in previous sections). Hence, the existing DHT routing techniques need to be augmented in order to efficiently resolve multidimensional queries. Various data structures that we discussed in the previous section effectively create a logical multidimensional index space over a DHT overlay. A look-up operation involves searching for an index or set of indexes in a multidimensional space. However, the exact query routing path in the multidimensional logical space is directly governed by the data distribution mechanism (i.e. based on the data structure that maintains the indexes). In this context, various approaches have proposed different routing/indexing heuristics.

Efficient query routing algorithms should exhibit the following characteristics [29]: (i) routing load balance: every peer in the network should route forward/route approximately the same number of query messages; and (ii) low routing state per Cloud peer: each Cloud peer should maintain a small number of routing links hence limiting new Cloud peer (VM) join and Cloud peer (VM) state update cost. In the current peer-to-peer literature, multidimensional data distribution mechanisms based on the following structures have been proposed: (i) space filling curves; and (ii) tree-based structures. Resolving multidimensional queries over a DHT overlay that utilizes SFCs for data distribution consists of two basic steps [10]: (i) mapping the multidimensional query onto the set of relevant clusters of SFC-based index space; and (ii) routing the message to all VMs that fall under the computed SFC-based index space. On the other hand, routing multidimensional query in a DHT overlay that employs tree-based structures for data distribution requires routing to start from the root. However, the root VM presents a single point of failure and load imbalance. To overcome this, the authors introduced the concept of fundamental minimum level. This means that all the query processing and the data storage should start at that minimal level of the tree rather than at the root. There are a number of techniques available for distributed routing in multidimensional space. The performance of techniques varies depending on the distribution of data in the multidimensional space, and VM in the underlying DHT overlay.

12.5 Cloud Peer Software Fabric: Design and Implementation

The Cloud peer implements services for enabling decentralized and distributed discovery supporting status look-ups and updates across the internetworked Cloud computing systems, enabling inter-application service co-ordinated provisioning for optimizing load-balancing and tackling the distributed service contention problem. The dotted box in Fig. 12.1 shows the layered design of Cloud peer service over DHT based self-organizing routing structure. The services built on the DHT routing structure extends (both algorithmically and programmatically) the fundamental properties related to DHTs including deterministic look-up, scalable routing, and decentralized network management. The Cloud peer service is divided into a number of sublayers (see Fig. 12.1): (i) higher level services for discovery, co-ordination, 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.

12.5.1 Overlay Construction

The overlay construction refers to how Cloud peers are logically connected over the physical network. The software implementation utilizes (the open source implementation of Pastry DHT known as the FreePastry) Pastry [32] as the basis for creation of Cloud peer overlay. A Pastry overlay interconnects 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 IS and query (discovery, update) ids, called keys, using cryptographic/randomizing hash functions such as SHA-1. These IDs are generated from 160-bit unique identifier space. The ID is used to indicate a Cloud peer's 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 (ii) building up routing information (leaf set, routing table, and neighborhood set) at various Cloud peers in the network. Given the Key K, the 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 to the proximity metric. Figure 12.2 shows a hypothetical Pastry overlay with keys and Cloud peers distributed on the circular ring based on their cryptographically generated IDs.

12.5.2 Multidimensional Query Indexing

To support multidimensional query indexing (Cloud service type, Host utilization, Instance OS type, Host Cloud location, Host Processor speed) over Pastry overlay, a Cloud peer implements a distributed indexing technique [33], which is a variant of peer-to-peer MX-CIF Quad tree [34] data structure. The distributed index builds a multidimensional attribute space based on the Cloud service attributes, where each attribute represents a single dimension. An example of a two-dimensional attribute space that indexes service attributes including speed and CPU type is shown in Fig. 12.2. The first step in initializing the distributed index is the process called minimum division (f_{min}). This process divides the Cartesian space into multiple index cells when the multidimensional 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 co-ordinate 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. 12.2, $f_{min} = 1$, dim = 2. The Pastry overlay hashing method (DHash (co-ordinates)) 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. 12.2, DHash(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.

12.5.3 Multidimensional 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 a mapping 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 multidimensional point service discovery query, the mapping is straightforward since every point is mapped to only one cell in the attribute space. For a multidimensional range query, 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. This mapping involves feeding the service discovery query's spatial dimensions into a mapping function, *IMap*(query).

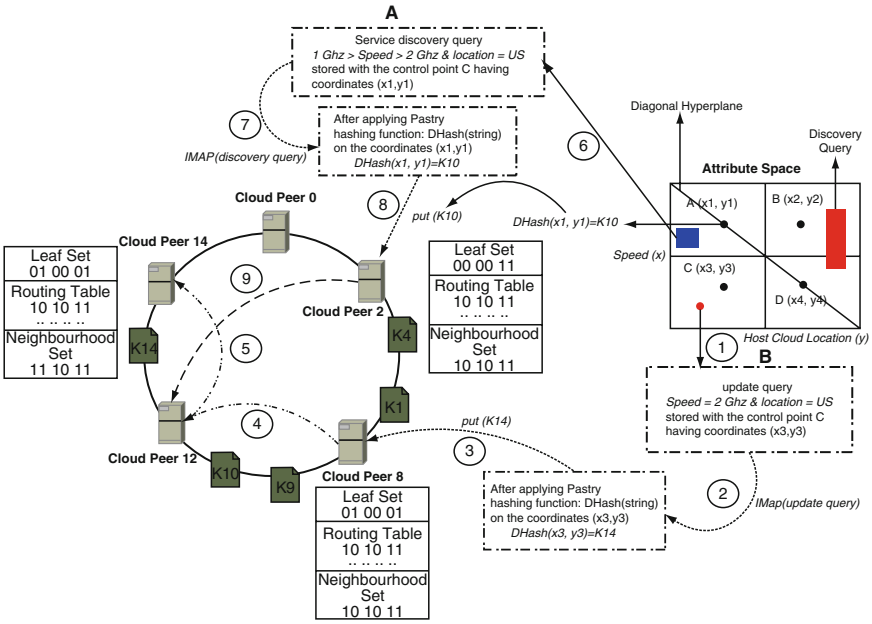

Fig. 12.2 A pictorial representation of Pastry (DHT) overlay construction, multidimensional data indexing, and routing: (1) a service hosted within a VM publishes an update query; (2) Cloud peer 8 computes the index cell, $C(x_3,y_3)$, to which the update query maps by using mapping function *IMap*(query); (3) next, distributed hashing function, *DHash*(x_3, y_3), is applied on the cell's co-ordinate values, which yields an 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) *DHash*(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

This function returns the IDs of index cells to which given query can be mapped (refer to step 7 in Fig. 12.2). Distributed hashing (*DHash*(cells)) 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 the update query also involves the identification of the cell in the attribute space using the same algorithm. An update query is always associated with an event region [35] 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 on the diagonal hyperplane of the attribute space. Giving in-depth information here is out of the scope for this chapter; however, the readers who would like to have more information can refer the paper [15, 30, 33] that describes the index in detail.

12.5.4 Designing Decentralized and Co-ordinated Load-Balancing Mechanism

A co-ordinated provisioning of requests between virtual machine instances deployed in Clouds is critical, as it prevents the service provisioners 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 multidimensional indexing technique is able to perform a decentralized and co-ordinated balancing of service provisioning requests among the set of available VMs. The description of the actual load-balancing mechanism follows next.

As mentioned in previous section, both service discovery query (issued by service provisioner) and update query (published by VMs or Services hosted within VMs) are spatially hashed to an index cell *i* in the multidimensional attribute space. In Fig. 12.3, a service discovery query for provisioning request P1 is mapped to an 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. 12.3, 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 IMap(query) function to an Cloud peer s. In this case, Cloud peer *s* is responsible for co-ordinating the provisioning of services among all the service discovery queries that are currently mapped to the cell *i*. Subsequently, VM *u* issues a resource ticket (see Fig. 12.3) that falls under a region of the attribute space currently required by the provisioning 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 VM *u*. The load-balancing decision is based on the principle that it should not lead to over-provisioning of service(s) hosted within VM *u*. This mechanism leads to co-ordinated load-balancing across VMs in Clouds and aids in achieving system-wide objective function.

The examples in Table 12.5 are service discovery queries that are stored with a Cloud peer service at time $T = 700$ s. Essentially, the queries in the list arrived at a time ≤ 700 and waited for a suitable update query that could meet their provisioning requirements (software, hardware, service type, location). Table 12.6 depicts an update query that arrived at $T = 700$. Following the update query arrival, the Cloud peer service undertakes a procedure that allocates the available service capacity with VM (that published the update query) among the list of matching service discovery queries. Based on the updating VM's attribute specification, only service discovery query 3 matches. Following this, the Cloud peer notifies the provisioning services that posted the query 3. Note that queries 1 and 2 have to wait for the arrival of update queries that can match their requirements.

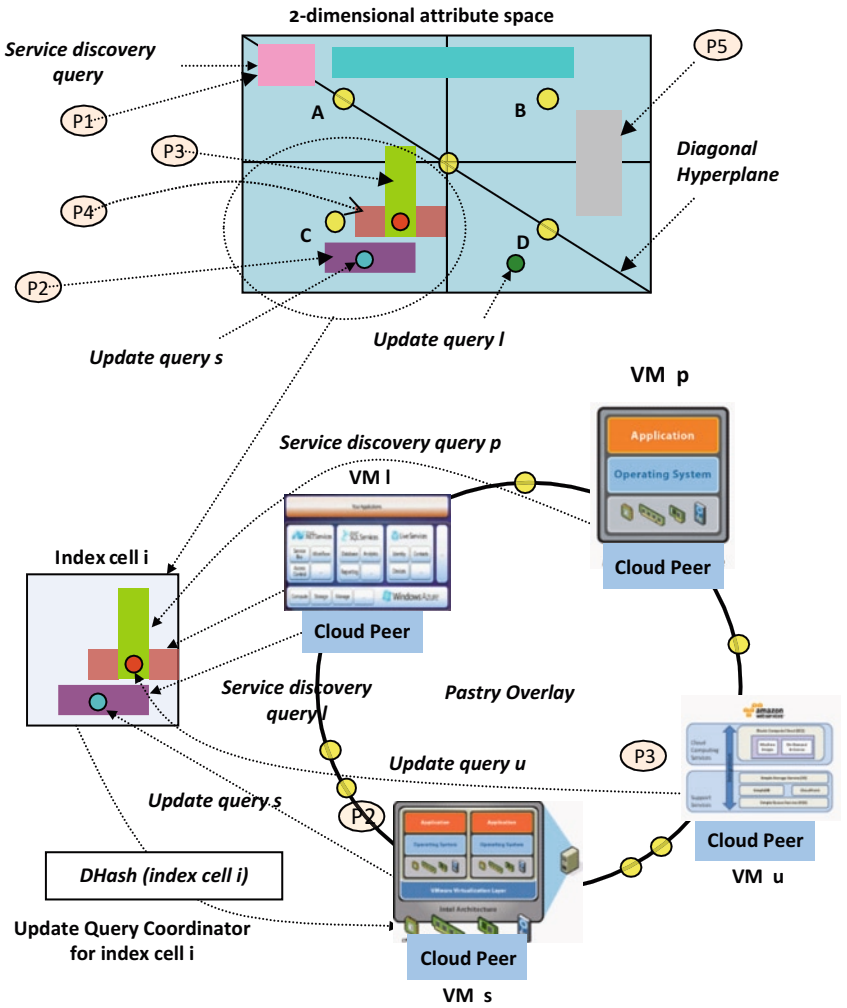

Fig. 12.3 Co-ordinated provisioning across VM instances: multidimensional service provisioning requests {P1, P2, P3, P4}, index cell control points {A, B, C, D}, multidimensional update queries {l, s}, and some of the spatial hashing to the Pastry overlay, i.e. the multidimensional (spatial) coordinate values of a cell's control point is used as the Pastry key. For this figure, f_{min} =2, dim = 2

Table 12.5 Service discovery query stored with a Cloud Peer service at time *T*

Time	Discovery query ID	Service type	Speed (GHz)	Cores	Location
300	Query 1	Web hosting	>2	1	USA
400	Query 2	Scientific simulation	>2	1	Singapore
500	Query 3	Credit card authenticator	>2.4	1	Europe

Table 12.6 Update query published with a Cloud Peer service at time *T*

Time	VM ID	Service type	Speed (GHz)	Processors	Type
700	VM 2	Credit card authenticator	2.7	One (available)	Europe

12.6 Experiments and Evaluation

In this section, we evaluate the performance of the proposed peer-to-peer Cloud provisioning concept by creating a service and VM pool that consists of multiple virtual machines that are hosted within the Amazon EC2 infrastructure. We assume unsaturated server availability for these experiments, so that enough capacity can 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), PaaS layer provisioning software, and application characteristics related to this experimental evaluation.

12.6.1 Cloud Peer Details

A Cloud peer service operates at PaaS 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 co-ordinated interactions and service selections. Every VM instance, which is deployed on a Cloud platform, hosts a Cloud peer service (see Figs. 12.2 and 12.3) that loosely glues it to the overlay. Next follows the details related to Cloud peer configuration.

FreePastry[1] *Network Configuration*: Both Cloud Peers' nodeIDs and discovery/update queries' IDs are randomly generated from and uniformly distributed in the 160-bit Pastry identifier space. Every Cloud peer service is configured to buffer a maximum of 1,000 messages at a given instance of time. The buffer size is chosen to be sufficiently large such that the FreePastry does not drop any messages. Other network parameters are configured to the default values as given in the file freepastry.params. This file is provided with the FreePastry distribution.

Multidimensional Index Configuration: The minimum division f_{min} of logical multidimensional index is set to 3, while the maximum height f_{max} of the distributed index tree is constrained to 3. In other words, the division of the multidimensional attribute space is not allowed beyond f_{min} for simplicity. The index space has provision for defining service discovery and update queries that specify the VM characteristics in four dimensions including number of application service type being hosted, 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.

[1] An open source pastry DHT implementation. http://freepastry.rice.edu/FreePastry

Service Discovery and Update Query's Multidimensional Extent: Update queries, which are posted by VM instances, express equality constraints on service, installed software environments, and hardware configuration attribute values (e.g. =).

12.6.2 Aneka: PaaS Layer Application Provisioning and Management Service

At PaaS layer, we utilize the Aneka [14] 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 system using Aneka, a developer or application scientist needs to start an instance of the configurable Aneka container hosting required services on each selected VM.

Services of Aneka can be clearly characterized into two distinct spaces: (i) Application Provisioner, a service that implements the functionality that accepts application workload from Cloud users, performs dynamic discovery of application management services via the Cloud peer service, dispatches workload to application management service, monitors the progress of their execution, and 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 end-point 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, a service, hosted within a VM, which is responsible for handling execution and management of submitted application workloads. An application management service sits within a VM and updates its usage status, software, and hardware configurations by sending update queries to the overlay. One or more instance of application management service can be connected in a single-level hierarchy to be controlled by a root-level Aneka Management Co-ordinator. This kind of service integration is aimed at making application programming flexible, efficient, and scalable.

12.6.3 Test Application

The PaaS layer software service, Aneka, supports composition and execution of application programs that are composed using different service models to be executed within the same software environment. The experimental evaluation in this chapter considers execution of applications programmed using a multithreaded programming model. The Thread programming model [14] 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 better for implementing and

architecting new applications and 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 on peer-to-peer network models, 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 cloud enables the Mandelbrot fractal calculation using a multithreaded programming model. The application submission interface allows the user to configure a number of horizontal and vertical partitions into which the fractal computation can be divided. The number of independent thread units created is equal to the horizontal x vertical partitions. For evaluations, we vary the values for horizontal and vertical parameters over the interval 5×5, 10×10, and 15×15. This configuration results in observation points.

12.6.4 Deployment of Test Services on Amazon EC2 Platform

To test the feasibility of the aforementioned services with regard to the provisioning of application services on Amazon EC2 cloud platform, we created Amazon Machine Images (AMIs) packaged with a Cloud peer, Application Management, and Aneka Management Co-ordinator services. The image that hosts the Aneka Management Co-ordinator is equipped with Microsoft Windows Server 2003 R2 Datacenter edition, Microsoft SQL Server 2005 Express, and Internet Information Services 6, while the AMI hosts only the Management Service and has Microsoft Windows Server 2003 R2 Datacenter system installed. For every AMI, we installed only the essential software including mandatory Cloud peer service, which is hosted within a Tomcat 6.0.10, Axis2 1.2 container. Cloud peer is exposed to the provisioning and management services through WS* interfaces. Later, we built our customized Amazon Machine Images from the two instances, creating and starting up more management co-ordinator and application management services by using customized images. We configured three management co-ordinators and nine management services. The management service is divided into groups of three that connect with a single co-ordinator resulting in a hierarchical structure. The management co-ordinator services communicate and internetwork through the Cloud peer fabric service. Figure 12.4 shows the pictorial representation of the experiment setup.

12.7 Results and Discussions

To measure the performance of peer-to-peer Cloud provisioning technique in regard to response time, co-ordination delay, and Pastry overlay network message complexity, we consider simultaneous provisioning of test applications at Application Provisioner A and B (see Table 12.7). The *response time* for an application is calculated by subtracting the output arrival time of the last thread in the submission

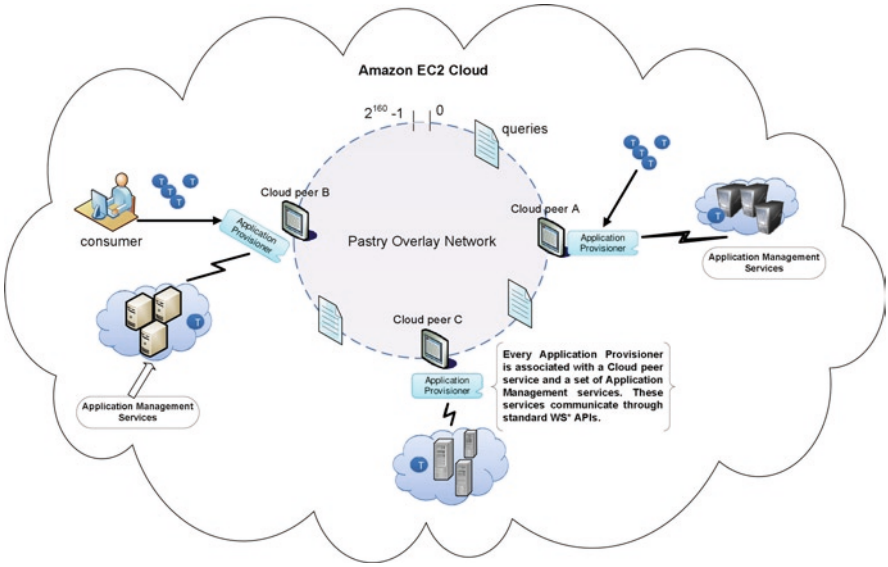

Fig. 12.4 Experiment Setup in Amazon EC2

Table 12.7 Response time, co-ordination delay, message count versus complexity

Problem complexity	5 × 5		10 × 10		15 × 15	
Provisioner	A	B	A	B	A	B
Response time (s)	27	27	107	104	245	229
Coordination delay (s)	5.58	7.13	26.08	24.97	60.06	48.09
Update message	3,203		3,668		3,622	
Discovery message	75		400		450	
Total message count	5,760		7,924		8,006	

list from the time at which the application is submitted. The metric *co-ordination delay* sums up the latencies for: (i) a service discovery query to be mapped to a Cloud peer, (ii) waiting time till an update query matches the discovery query; and (ii) notification delay from the Cloud peer to the Application Provisioner that originally posted the service discovery query. Pastry overlay message complexity measures the details related to the 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 overlay, and (iv) send notifications.

Table 12.7 (response time vs. complexity) shows the results for response time in seconds with increasing complexity/problem size for the test application. Cloud consumers submit their applications with provisioner A and B. The initial experimental results reveal that with increase in problem complexity (number of horizontal and vertical partitions), the Cloud consumers experience increase in response times. The basic reason behind this behaviour of the provisioning system is related

to the fixed number Application Management services, i.e. 9, available to the Application Provisioners. With increase in the problem complexity, the number of job threads (a job thread represents a single work unit, e.g. for a 5 × 5 Mandelbrot configuration, 25 job threads are created and submitted with the Application Provisioner) that are to be executed with management services increase, hence leading to worsening queuing and waiting delays. However, this behaviour of the provisioning system can be fixed through implementation of reactive provisioning of new VM instances to reflect the sudden surge in application workload processing demands (problem complexity). In our future work, we want to explore how to dynamically provision or de-provision VMs and associated Application Management services driven by workload processing demands.

Table 12.7 (coordination delay vs. complexity) presents the measurements for average co-ordination 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 co-ordination delay. This happens because the discovery queries of the corresponding job threads have to wait for a longer period of time before they are matched against an update query object. However, the job thread processing time (CPU time) is not affected by the co-ordination delay; hence, the response time in Table 12.7 shows a similar trend to delay.

In Table 12.7 (message count vs. complexity), we show the message overhead involved with management of multidimensional index, routing of discovery and update query messages, and maintenance of Pastry overlay. We can clearly see that as application size (problem complexity) increase, the number of messages required for mapping the query objects and maintenance of the overlay network increase. The number of discovery and update messages produced in the overlay is a function of the multidimensional index structure that indexes and routes these queries in a distributed fashion. Hence, the choice of the multidimensional data indexing structure and routing technique governs the manageability and efficiency of the overlay network (latency, 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.

12.8 Conclusions and Path Forward

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, they can gracefully adapt to the dynamic system expansion (join) or contraction (leave, failure), and are not susceptible to a 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 confirm 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) and processing load, preserve data locality, and minimize the metadata. Another important algorithmic and programming challenge in building robust Cloud peer services is to guarantee consistent routing, look-up, 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.

References

1. Armbrust M, Fox A, Griffith R, Joseph A, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2009) Above the clouds: a Berkeley view of cloud computing. University of California at Berkley, USA. Technical Rep UCB/EECS-2009-28
2. The Reservoir Seed Team (2008) Reservoir – an ICT infrastructure for reliable and effective delivery of services as utilities. IBM Res Rep H-0262
3. Buyya R, Yeo C, 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 Gen Comput Syst 25:599–616
4. Google (2009) Google App Engine. https://appengine.google.com/. Accessed 16 Dec 2009
5. Ultra Serve Internet Pty Ltd (2009) Rejila On Demand Cloud Computing Servers. http://www.rejila.com/. Accessed 14 Dec 2009
6. Rackspace US, Inc. (2009) The Rackspace Cloud. http://www.rackspacecloud.com. Accessed 12 Dec 2009
7. Microsoft (2009) Windows Azure Platform, http://www.microsoft.com/windowsazure/. Accessed 12 Dec 2009
8. Amazon Web Services LLC (2009) Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/. Accessed 16 Dec 2009
9. Salesforce.com (2009) Application Development with Force.com's Cloud Computing Platform http://www.salesforce.com/platform/. Accessed 16 Dec 2009
10. Quiroz A, Kim H, Parashar M, Gnanasambandam N, Sharma N (2009) Towards autonomic workload provisioning for enterprise grids and clouds. In: Proceedings of the 10th IEEE/ACM international conference on grid computing, Banf, Alberta, Canada, 13–15 Oct 2009, IEEE Computer Society Press
11. Amazon Web Services LLC (2009) Amazon CloudWatch. http://aws.amazon.com/cloudwatch/. Accessed 22 Sept 2009
12. Amazon Web Services LLC (2009) Elastic Load Balancer http://aws.amazon.com/elasticloadbalancing/. Accessed 22 Sept 2009
13. Rochwerger B, Breitgand D, Levy E, Galis A, Nagin K, Llorente L, 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 Syst J 53
14. Chu X et al (2007) Aneka: next-generation enterprise grid platform for e-Science and e-Business applications. In: Proceedings of the 3rd IEEE international conference on e-Science and grid computing, Bangalore, India

15. Ranjan R, Chan L, Harwood A, Karunasekera S, Buyya R (2007) Decentralised resource discovery service for large scale federated grids. In: Proceedings of the 3rd IEEE international conference on eScience and grid computing (eScience'07), Bangalore, India, IEEE Computer Society, Los Alamitos, CA
16. Eucalyptus Systems Inc (2009) Eucalyptus Systems. http://www.eucalyptus.com/. Accessed 22 Sept 2009
17. Amazon Web Services LLC (2009) Auto Scaling. http://aws.amazon.com/autoscaling/. Accessed 22 Sept 2009
18. GoGrid Cloud Hosting (2009) F5 Load Balancer. GoGrid Wiki. http://wiki.gogrid.com/wiki/index.php/(F5)_Load_Balancer. Accessed 21 Sept 2009
19. Bakhtiari S, Safavi-Naini R, Pieprzyk J (1995) Cryptographic Hash Functions: A Survey. http://citeseer.ist.psu.edu/bakhtiari95cryptographic.html. Accessed 22 Sept 2009
20. Balakrishnan H, Kaashoek MF, Karger D, Morris R, Stoica I (2003) Looking up data in peer-to-peer systems. Commun ACM 46(2):43–48
21. 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: Proceedings of the 29th annual ACM symposium on theory of computing (STOC '97), New York. ACM Press, pp 654–663
22. Preneel B (1999) The state of cryptographic hash functions. In: Lectures on data security. Modern cryptology in theory and practice. Springer, London, pp 158–182
23. Lua K, Crowcroft J, Pias M, Sharma R, Lim S (2005) A survey and comparison of peer-to-peer overlay network schemes. IEEE Commun Surv Tutorials 7(2), IEEE Communications Society Press, Washington DC, USA
24. Li J, Stribling J, Gil TM, Morris R, Frans Kaashoek M (2004) Comparing the performance of distributed hash tables under churn. In: Proceedings of the 3rd international workshop on peer-to-peer systems (IPTPS04), San Diego, CA
25. Bharambe A, Agarwal M, Seshan S (2004) Mercury: supporting scalable multi-attribute range queries. In: Proceedings of SIGCOMM 2004 (SIGCOMM'04). ACM, Portland, OR
26. Castro M, Costa M, Rowstron A (2004) Should we build Gnutella on a structured overlay? SIGCOMM Comput Commun Rev 34(1):131–136
27. Linga P, Demers A, Gupta I, Birman K, van R (2003) Kelips: building an efficient and stable peer-to-peer DHT through increased memory and background overhead. In: Proceedings of the 2nd international workshop on peer-to-peer systems (IPTPS03), Berkeley, CA
28. Spence D, Crowcroft J, Hand S, Harris T (2005) Location based placement of Whole Distributed Systems. In: Proceedings of the 2005 ACM conference on Emerging network experiment and technology (CoNEXT'05). ACM Press, New York, pp 124–134
29. Ganesan P, Yang B, Garcia-Molina H (2004) One torus to rule them all: multi-dimensional queries in peer-to-peer systems. In: Proceedings of the 7th International Workshop on the Web and Databases (WebDB '04). ACM Press, New York, pp 19–24
30. Ranjan R, Harwood A, Buyya R (2008) Peer-to-peer based resource discovery in global grids: a tutorial. IEEE Commun Surv Tutorials 10(2):6–33
31. Samet H (1989) The design and analysis of spatial data structures. Addison–Wesley, Reading, MA
32. Rowstron A, Druschel P (2001) Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: IFIP/ACM international conference on distributed system platforms, Heidelberg
33. Ranjan R (2007) Coordinated resource provisioning in federated grids. Ph.D. thesis, The University of Melbourne
34. Tanin E, Harwood A, Samet H (2007) Using a distributed quadtree index in peer-to-peer networks. VLDB J 16(2):165–178, Springer, New York
35. Gupta A, Sahin OD, Agrawal D, Abbadi AEl (2004) Meghdoot: content-based publish/subscribe over peer-to-peer networks. In: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (Middleware '04), New York. Springer, New York, pp 254–273