Chapter 10

CometCloud: An Autonomic Cloud Engine

Hyunjoo Kim and Manish Parashar

1.1 Introduction

Clouds typically have highly dynamic demands for resources with highly heterogeneous and dynamic workloads. For example, the workloads associated with the application can be quite dynamic, both in terms of the number of tasks processed as well as computation requirements of each task. Furthermore, different applications may have very different and dynamic QoS requirements, for example, one application may require high throughput while another may be constrained by a budget, and a third may have to balance both throughput and budget. The performance of a cloud service can also vary based on these varying loads as well as failures, network conditions, etc., resulting in different quality of service to the application.

Combining public cloud platforms and integrating them with existing grids and data centers can support on-demand scale-up, scale-down as well as scale-out. Users may want to use resources in their private cloud (or datacenter or grid) first before scaling out onto a public cloud, and may have preference for a particular cloud or may want to combine multiple clouds. However, such integration and interoperability is currently non-trivial. Furthermore, integrating these public cloud platforms with exiting

computational grids provides opportunities for on-demand scale-up and scale-down, i.e., cloudbursts.

In this chapter, we present the CometCloud autonomic cloud engine. The overarching goal of CometCloud is to realize a virtual computational cloud with resizable computing capability, which integrates local computational environments and public cloud services on-demand, and provide abstractions and mechanisms to support a range of programming paradigms and applications requirements. Specifically, CometCloud enables policy-based autonomic *cloudbridging* and *cloudbursting*. Autonomic cloudbridging enables on-the-fly integration of local computational environments (datacenters, grids) and public cloud services (such as Amazon EC2 and Eucalyptus), and autonomic cloudbursting enables dynamic application scale-out to address dynamic workloads, spikes in demands, and other extreme requirements.

CometCloud is based on a decentralized coordination substrate, and supports highly heterogeneous and dynamic cloud/Grid infrastructures, integration of public/private clouds and cloudbursts. The coordination substrate is also used to support a decentralized and scalable task space that coordinates the scheduling of task, submitted by a dynamic set of users, onto sets of dynamically provisioned workers on available private and/or public cloud resources based on their QoS constraints such as cost or performance. These QoS constraints along with policies, performance history and the state of resources are used to determine the appropriate size and mix of the public and private clouds that should be allocated to a specific application request.

This chapter also demonstrates the ability of CometCloud to support the dynamic requirements of real applications (and multiple application groups) with varied computational requirements and QoS constraints. Specifically, this chapter describes two

applications enabled by CometCloud, a computationally intensive Value at Risk (VaR) application and a high-throughput medical image registration. VaR is a market standard risk measure used by senior managers and regulators to quantify the risk level of a firm's holdings. A VaR calculation should be completed within the limited time and the computational requirements for the calculation can change significantly. Image registration is the process to determine the linear/nonlinear mapping between two images of the same object or similar objects. In image registration, a set of image registration methods are used by different (geographically distributed) research groups to process their locally stored data. The set of images will be typically acquired at different time, or from different perspectives, and will be in different coordinate systems. It is therefore critical to align those images into the same coordinate system before applying any image analysis.

The rest of this chapter is organized as follows. We present the CometCloud architecture in section 1.2. Section 1.3 elaborates policy-driven autonomic cloudbursts, specifically, autonomic cloudbursts for real-world applications, autonomic cloudbridging over a virtual cloud, and runtime behavior of CometCloud. Section 1.4 states the overview of VaR and image registration applications. We evaluate the autonomic behavior of CometCloud in section 1.5 and conclude this paper in section 1.6.

1.2 CometCloud Architecture

CometCloud is an autonomic computing engine for cloud and grid environments. It is based on the Comet [5] decentralized coordination substrate, and supports highly heterogeneous and dynamic cloud/Grid infrastructures, integration of public/private

1.2.1 CometCloud Layered Abstractions

A schematic overview of the CometCloud architecture is presented in Figure 1.1. The infrastructure layer uses the Chord self-organizing overlay [6], and the Squid [7] information discovery and content-based routing substrate built on top of Chord. The routing engine [8] supports flexible content-based routing and complex querying using partial keywords, wildcards, or ranges. It also guarantees that all peer nodes with data elements that match a query/message will be located. Nodes providing resources in the overlay have different roles and accordingly, different access privileges based on their

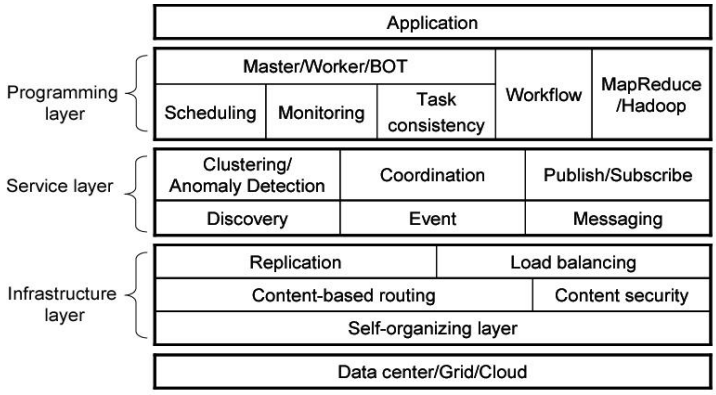


Figure 1.1: The CometCloud architecture for autonomic cloudbursts

The service layer provides a range of services to supports autonomics at the programming and application level. This layer supports the Linda-like [9] tuple space coordination model, and provides a virtual shared-space abstraction as well as associative access primitives. The basic coordination primitives are listed below:

- ***out*** *(ts, t)*: a non-blocking operation that inserts tuple *t* into space *ts*.
- ***in*** *(ts, t')*: a blocking operation that removes a tuple *t* matching template *t'* from the space *ts* and returns it.

- ***rd*** *(ts, t')*: a blocking operation that returns a tuple *t* matching template *t'* from the space *ts*. The tuple is not removed from the space.

The *out* for inserting a tuple into the space, *in* and *rd* for reading a tuple from the space are implemented. *in* removes the tuple after read and *rd* only reads the tuple. We support range query, hence '*' can be used for searching all tuples. The above uniform operators do not distinguish between local and remote spaces, and consequently the Comet is naturally suitable for context-transparent applications. However, this abstraction does not maintain geographic locality between peer nodes, and may have a detrimental effect on the efficiency of the applications imposing context-awareness, e.g., mobile applications. These applications require that context locality be maintained in addition to content locality, i.e., they impose requirements for context-awareness. To address this issue, CometCloud supports dynamically constructed transient spaces that have a specific scope definition (e.g., within the same geographical region or the same physical subnet). The global space is accessible to all peer nodes and acts as the default coordination platform. Membership and authentication mechanisms are adopted to restrict access to the transient spaces. The structure of the transient space is exactly the same as the global space. An application can switch between spaces at runtime and can simultaneously use multiple spaces. This layer also provides asynchronous (publish/subscribe) messaging and evening services. Finally, online clustering services support autonomic management and enable self-monitoring and control. Events describing the status or behavior of system components are clustered and the clustering is used to detect anomalous behaviors.

The programming layer provides the basic framework for application development and management. It supports a range of paradigms including the

1.2.2 Comet Space

In Comet, a tuple is a simple XML string, where the first element is the tuple's tag and is followed by an ordered list of elements containing the tuple's fields. Each field has a name followed by its value. The tag, field names, and values must be actual data for a tuple and can contain wildcards (“*”) for a template tuple. This lightweight format is flexible enough to represent the information for a wide range of applications, and can support rich matching relationships [21]. Further, the cross-platform nature of XML makes this format suitable for information exchange in distributed heterogeneous environments.

A tuple in Comet can be retrieved if it exactly or approximately matches a template tuple. Exact matching requires the tag and field names of the template tuple to be specified without any wildcard, as in Linda. However, this strict matching pattern

must be relaxed in highly dynamic environments, since applications (e.g., service discovery) may not know exact tuple structures. Comet supports tuple retrievals with incomplete structure information using approximate matching, which only requires the tag of the template tuple be specified using a keyword or a partial keyword. Examples are shown in Figure 1.2. In this figure, tuple (a) tagged "contact" has fields "name, phone, email, dep" with values "Smith, 7324451000, smith@gmail.com, ece", can be retrieved using tuple template (b) or (c).

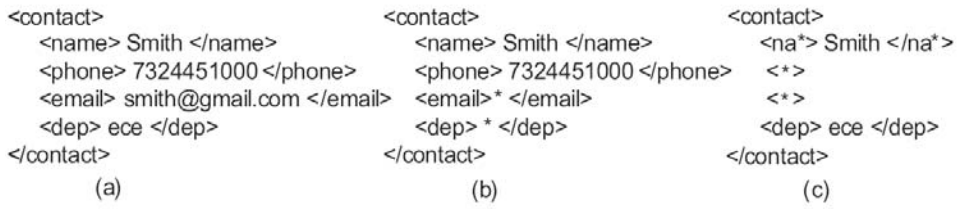


Figure 1.2: Example of tuples in CometCloud

Comet adapts Squid information discovery scheme and employs the Hilbert Space-Filling Curve (SFC) [22] to map tuples from a semantic information space to a linear node index. The semantic information space, consisting of based-10 numbers and English words, is defined by application users. For example, a computational storage resource may belong to the 3D storage space with coordinates "space", "bandwidth", and "cost". Each tuple is associated with k keywords selected from its tag and field names, which are the keys of a tuple. For example, the keys of tuple (a) in Figure 1 can be "name, phone" in a 2D student information space. Tuples are local in the information space if their keys are lexicographically close, or if they have common keywords. The selection of keys can be specified by the applications.

A Hilbert SFC is a locality preserving continuous mapping from a k-dimensional (kD) space to a 1D space. It is locality preserving in that points that are close on the curve

are mapped from close points in the kD space. The Hilbert curve readily extends to any number of dimensions. Its locality preserving property enables the tuple space to maintain content locality in the index space. In Comet, the peer nodes form a 1-dimensional overlay, which is indexed by a Hilbert SFC. Applying the Hilbert mapping, the tuples are mapped from the multi-dimensional information space to the linear peer index space. As a result, Comet uses the Hilbert SFC constructs the distribute hash table (DHT) for tuple distribution and lookup. If the keys of a tuple only include complete keywords, the tuple is mapped as a point in the information space and located on at most one node. If its keys consist of partial keywords, wildcards, or ranges, the tuple identifies a region in the information space. This region is mapped to a collection of segments on the SFC and corresponds to a set of points in the index space. Each node stores the keys that map to the segment of the curve between itself and the predecessor node. For example, as shown in Figure 1.3, five nodes (with id shown in solid circle) are indexed using SFC from 0 to 63, the tuple defined as the point (2, 1) is mapped to index 7 on the SFC and corresponds to node 13, and the tuple defined as the region (2-3, 1-5) is mapped to 2 segments on the SFC and corresponds to nodes 13 and 32.

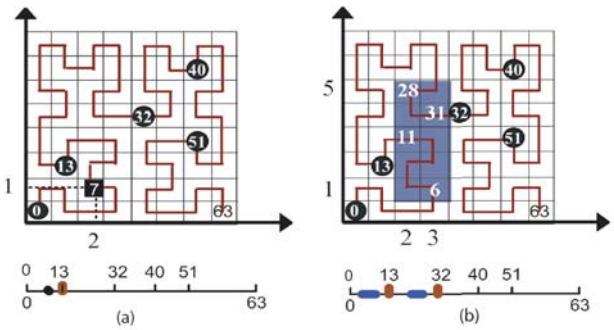


Figure 1.3: Examples of mapping tuples from 2D information space to 1D index space [5]

1.3 Autonomic Behavior of CometCloud

1.3.1 Autonomic Cloudbursting

The goal of autonomic cloudbursts is to seamlessly and securely integrate private enterprise clouds and datacenters with public utility clouds on-demand, to provide the abstraction of resizable computing capacity. It enables the dynamic deployment of application components, which typically runs on internal organizational compute resources, onto a public cloud to address dynamic workloads, spikes in demands, and other extreme requirements. Furthermore, given the increasing application and infrastructure scales, as well as their cooling, operation and management costs, typical over-provisioning strategies are no longer feasible. Autonomic cloudbursts can leverage utility clouds to provide on-demand scale-out and scale-in capabilities based on a range of metrics.

The overall approach for supporting autonomic cloudbursts in CometCloud is presented in Figure 1.4. CometCloud considers three types of clouds based on perceived security/trust and assigns capabilities accordingly. The first is a highly trusted, robust and secure cloud, usually composed of trusted/secure nodes within an enterprise, which is typically used to host masters and other key (management, scheduling, monitoring) roles. These nodes are also used to store states. In most applications, the privacy and integrity of critical data must be maintained, and as a result, tasks involving critical data should be limited to cloud nodes that have required credentials. The second type of cloud is one composed of nodes with such credentials, i.e., the cloud of secure workers. A privileged Comet space may span these two clouds and may contain critical data, tasks and other

aspects of the application-logic/workflow. The final type of cloud consists of casual workers. These workers are not part of the space but can access the space through the proxy and a request handler to obtain (possibly encrypted) work units as long as they present required credentials. Nodes can be added or deleted from any of these clouds by purpose. If the space needs to be scale-up to store dynamically growing workload as well as requires more computing capability, then autonomic cloudbursts target secure worker to scale up. But only if more computing capability is required, then unsecured workers are added.

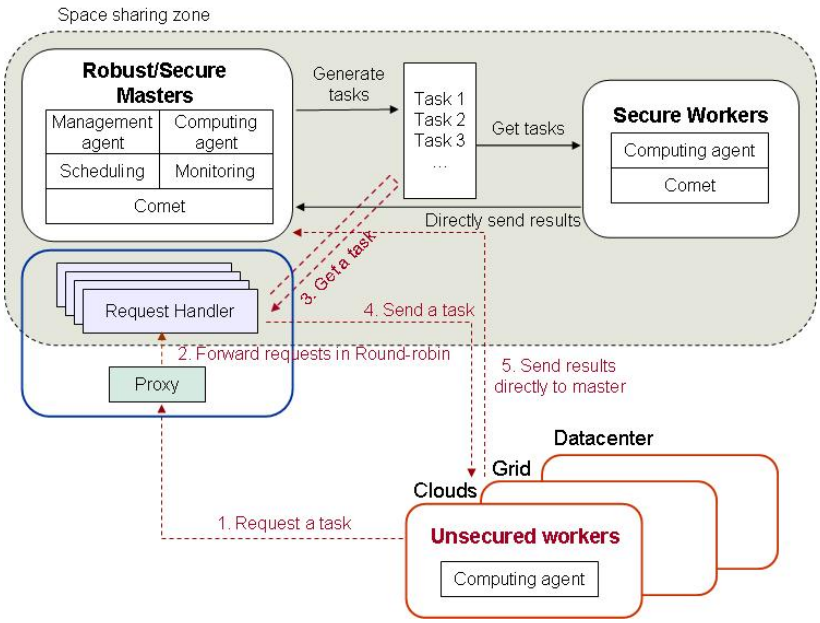


Figure 1.4: Autonomic cloudbursts using CometCloud

Key motivations for autonomic cloudbursts include:

- *Load dynamics*: Application workloads can vary significantly. This includes the number of application tasks as well the computational requirements of a task. The computational environment must dynamically grow (or shrink) in response to these dynamics while still maintaining strict deadlines.

- *Accuracy of the analytics*: The required accuracy of risk analytics depends on a number of highly dynamic market parameters, and has a direct impact on the computational demand, e.g., the number of scenarios in the Monte-Carlo VaR formulation. The computational environment must be able to dynamically adapt to satisfy the accuracy requirements while still maintaining strict deadlines.
- *Collaboration of different groups*: Different groups can run the same application with different data sets policies. Here, policy means user's SLA bounded by their condition such as time frame, budgets and economic models. As collaboration groups join or leave the work, the computational environment must grow or shrink to satisfy their SLA.
- *Economics*: Application tasks can have very heterogeneous and dynamic priorities, and must be assigned resources and scheduled accordingly. Budgets and economic models can be used to dynamically provision computational resources based on the priority and criticality of the application task. For example, application tasks can be assigned budgets and can be assigned resources based on this budget. The computational environment must be able to handle heterogeneous and dynamic provisioning and scheduling requirements.
- *Failures*: Due to the strict deadlines involved, failures can be disastrous. The computation must be able to manage failures without impacting application quality of service, including deadlines and accuracies.

1.3.2 Autonomic Cloudbridging

Autonomic cloudbridging is meant to connect CometCloud and a virtual cloud which consists of public cloud, datacenter and grid by the dynamic needs of the application. The clouds in the virtual cloud are heterogeneous and have different types of resources and cost policies, besides, the performance of each cloud can change over time by the number of current users. Hence, types of used clouds, the number of nodes in each cloud and resource types of nodes should be decided according to the changing environment of the clouds and application's resource requirements.

Figure 1.5 shows an overview of the operation of the CometCloud-based autonomic cloudbridging. The scheduling agent manages autonomic cloudbursts over the virtual cloud and there can be one or more scheduling agents. A scheduling agent is located at robust/secure master site. If multiple collaborating research groups work together and each group requires generating tasks with its own data and managing the virtual cloud by its own policy, then it can have a separate scheduling agent in its master site. The requests for tasks generated by the different sites are logged in the CometCloud virtual shared space that spans master nodes at each of the sites. These tasks are then consumed by workers, which may run on local computational nodes at the site, a shared datacenter, a grid or on a public cloud infrastructure. A scheduling agent manages QoS constraints and autonomic cloudbursts of its site according to the defined policy. The workers can access the space using appropriate credentials, access authorized tasks and return results back to the appropriate master indicated in the task itself.

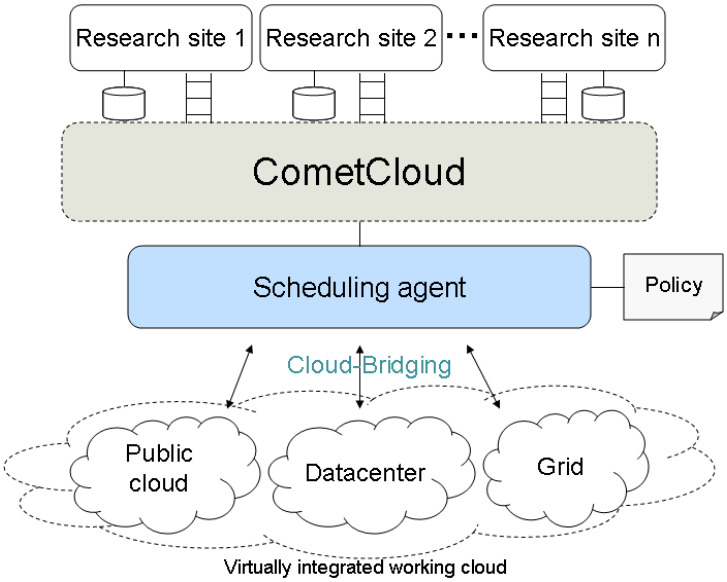


Figure 1.5: Overview of the operation of autonomic cloudbridging

A scheduling agent manages autonomic cloudbridging and guarantees QoS within user policies. Autonomic cloudburst is represented by changing resource provisioning not to violate defined policy. We define three types of policies.

- *Deadline-based*: When an application needs to be completed as soon as possible, assuming an adequate budget, the maximum required workers are allocated for the job.
- *Budget-based*: When a budget is enforced on the application, the number of workers allocated must ensure that the budget is not violated.
- *Workload-based*: When the application workload changes, the number of workers explicitly defined by the application is allocated or released.

1.3.3 Other Autonomic Behaviors

Fault-tolerance

Supporting fault-tolerance during runtime is critical to keep the application's deadline. We support fault-tolerance in two ways which are in the infrastructure layer and in the programming layer. The replication substrate in the infrastructure layer provides a mechanism to keep the same state as that of its successor's state, specifically coordination space and overlay information. Figure 1.6 shows the overview of replication in the overlay. Every node has a local space in the service layer and a replica space in the infrastructure layer. When a tuple is inserted or extracted from the local space, the node notifies this update to its predecessor and the predecessor updates the replica space. Hence every node keeps the same replica of its successor's local space. When a node fails, another node in the overlay detects the failure and notifies it to the predecessor of the failed node. Then the predecessor of the failed node merges the replica space into the local space and this makes all the tuples from the failed node recovered. Also the predecessor node makes a new replica for the local space of its new successor. We also support fault-tolerance in the programming layer. Even though replica of each node is maintained, some tasks can be lost during runtime because of network congestion or task generation during failure. To address this issue, the master checks the space periodically and regenerates lost tasks.

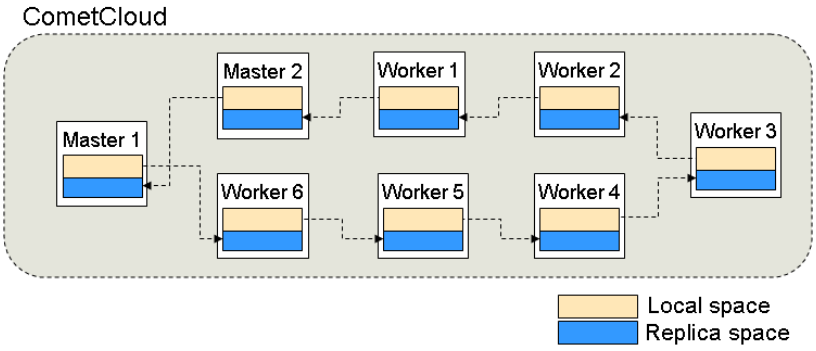


Figure 1.6: Replication overview in the CometCloud overlay

Load Balancing

In a cloud environment, executing application requests on underlying grid resources consists of two key steps. The first, which we call *VM Provisioning*, consists of creating VM instances to host each application request, matching the specific characteristics and requirements of the request. The second step is mapping and scheduling these requests onto distributed physical resources (*Resource Provisioning*). Most virtualized data centers currently provide a set of general-purpose VM classes with generic resource configurations, which quickly become insufficient to support the highly varied and interleaved workloads. For example, Amazon's EC2 only supports five basic VM types [1]. Furthermore, clients can easily under- or over-estimate their needs because of a lack of understanding of application requirements due to application complexity and/or uncertainty, and often results in over-provisioning due to a tendency to be conservative.

The decentralized clustering approach specifically addresses the distributed nature of enterprise grids and clouds. The approach builds on a decentralized messaging and data analysis infrastructure that provides monitoring and density-based clustering capabilities. By clustering workload requests across data center job queues, the characterization of different resource classes can be accomplished to provide autonomic

VM provisioning. This approach has several advantages, including the capability of analyzing jobs across a dynamic set of distributed queues, the non-dependency on a priori knowledge of the number of clustering classes, and the amenity for online application and timely adaptation to changing workloads and resources. Furthermore, the robust nature of the approach allows it to handle changes (joins/leaves) in the job queue servers as well as their failures while maximizing the quality and efficiency of the clustering.

1.4 Overview of CometCloud-based Applications

In this section, we describe two types of applications which are VaR for measuring the risk level of a firm's holdings and Image registration for medical informatics. A VaR calculation should be completed within the limited time and the computational requirements for the calculation can change significantly. Besides, the requirement for additional computation happens irregularly. Hence, for VaR we will focus on how autonomic cloudbursts work for dynamically changing workloads. Image registration is the process to determine the linear/nonlinear mapping T between two images of the same object or similar objects which acquired at different time, or from different perspectives. Besides, because a set of image registration methods are used by different (geographically distributed) research groups to process their locally stored data, jobs can be injected from multiple sites. Another distinguished difference between two applications is that data size of image registration is much larger than that of VaR. In case of 3D image, image size is usually a few tens of mega bytes. Hence, image data should be separated from its task tuple and instead, it locates on a separate storage server and its location is indicated in the task tuple. For image registration, because it usually needs to

be completed as soon as possible within budget limit, we will focus on how CometCloud works using budget-based policy.

1.4.1 Value at Risk (VaR)

Monte-Carlo VaR is a very powerful measure used to judge the risk of portfolios of financial instruments. The complexity of the VaR calculation stems from simulating portfolio returns. To accomplish this, Monte-Carlo methods are used to "guess" what the future state of the world may look like. Guessing a large number of times allows the technique to encompass the complex distributions and the correlations of different factors that drive portfolio returns into a discreet set of *scenarios*. Each of these Monte-Carlo scenarios contains a state of the world comprehensive enough to value all instruments in the portfolio, and thus allows us to calculate a return for the portfolio under that scenario.

The process of generating Monte-Carlo scenarios begins by selecting primitive instruments or *invariants*. To simplify simulation modeling, invariants are chosen such that they exhibit returns that can be modeled using a stationary normal probability distribution [13]. In practice these invariants are returns on stock prices, interest rates, foreign exchange rates, etc. The universe of invariants must be selected such that portfolio returns are driven only by changes to the invariants.

To properly capture the nonlinear pricing of portfolios containing options, we use Monte-Carlo techniques to simulate many realizations of the invariants. Each realization is referred to as a scenario. Under each of these scenarios, each option is priced using the invariants and the portfolio is valued. As outlined above, the portfolio returns for scenarios are ordered from worst loss to best gain, and a VaR number is calculated.

1.4.2 Image Registration

Nonlinear image registration [14] is the computationally expensive process to determine the mapping *T* between two images of the same object or similar objects acquired at different time, in different position or with different acquisition parameters or modalities. Both intensity/area based and landmark based methods have been reported to be effective in handling various registration tasks. Hybrid methods which integrate both techniques have demonstrated advantages in the literature [15][16][17].

Alternative landmark point detection and matching method are developed as a part of hybrid image registration algorithm for both 2D and 3D images [18]. The algorithm starts with automatic detection of a set of landmarks in both fixed and moving images, followed by a coarse to fine estimation of the nonlinear mapping using the landmarks. Intensity template matching is further used to obtain the point correspondence between landmarks in the fixed and moving images. Because there is a large portion of outliers in the initial landmark correspondence, a robust estimator, RANSAC [19], is applied to reject outliers. The final refined inliers are used to robustly estimate a Thin Spline Transform (TPS) [20] to complete the final nonlinear registration.

1.5 Implementation and Evaluation

In this section, we evaluate basic CometCloud operations first, and then compare application runtime varying the number of nodes after describing how the applications were implemented using CometCloud. Then we evaluate VaR using workload-based policy and Image registration using budget-based policy. Also we evaluate CometCloud with/without a scheduling agent. For deadline-based policy which doesn't have budget

limit, because it allocates as many workers as possible, we applied it just to compare results with and without scheduling agent for budget-based policy.

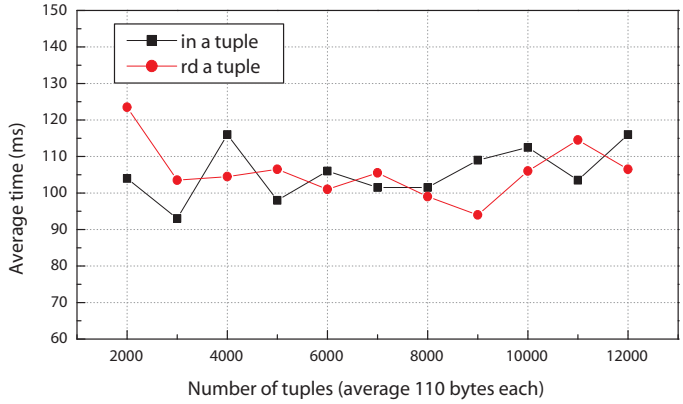
1.5.1 Evaluation of CometCloud

Basic CometCloud operations

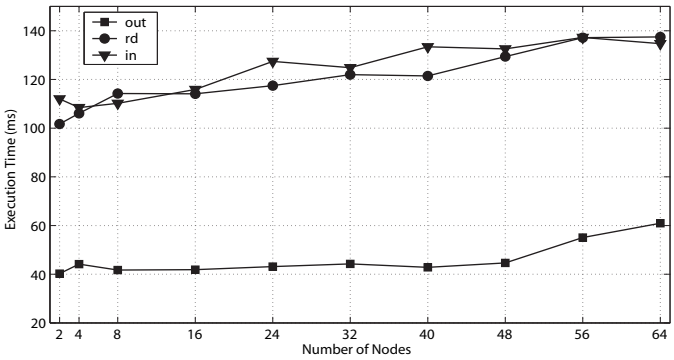
In this experiment we evaluated the costs of basic tuple insertion and exact retrieval operations on the Rutgers cloud. Each machine was a peer node in the CometCloud overlay and the machines formed a single CometCloud peer group. The size of the tuple in the experiment was fixed at 200 bytes. Ping-pong like process was used in the experiment, in which an application process inserted a tuple into the space using the *out* operator, read the same tuple using the *rd* operator, and deleted it using the *in* operator. In the experiment, the *out* and exact matching *in/rd* operators used a 3-dimension information space. For an *out* operation, the measured time corresponded to the time interval between when the tuple was posted into the space and when the response from the destination was received. For an *in* or *rd* operation, the measured time was the time interval between when the template was posted into the space and when the matching tuple was returned to the application, assuming that a matching tuple existed in the space. This time included the time for routing the template, matching tuples in the repository, and returning the matching tuple. The average performances were measured for different system sizes.

Figure 1.7 (a) plots the average measured performance and shows that the system scales well with increasing number of peer nodes. When the number of peer nodes increases 32 times, i.e., from 2 to 64, the average round trip time increases only about 1.5

times, due to the logarithmic complexity of the routing algorithm of the Chord overlay. *rd* and *in* operations exhibit similar performance, as shown in the figure. To further study the *in/rd* operator, the average time for *in/rd* was measured using increasing number of tuples. Figure 1.7 (b) shows that the performance of *in/rd* is largely independent of the number of tuples in the system – the average time is approximately 105 ms as the number of tuples is increased from 2,000 to 12,000.



(a) Average time for out, in, and rd operators for increasing system sizes



(b) Average time for in and rd operations with increasing number of tuples. System size fixed at 4 nodes

Figure 1.7: Evaluation of CometCloud primitives on the Rutgers cloud

Overlay join overhead

To share the Comet space, a node should join the CometCloud overlay and each node should manage a finger table to keep track of changing neighbors. When a node joins the overlay it first connects to a predefined bootstrap node and sends its information such as

IP address to the bootstrap. Then the bootstrap node makes a finger table for the node and sends it back to the node. Hence, the more nodes join the overlay at the same time, the larger join overhead happens. Table 1.1 shows the join overhead varying the number of joining nodes at the same time. We evaluated it on Amazon EC2 and the figure shows that the join overhead is less than 4 seconds even when 100 nodes join the overlay at the same time.

Table 1.1: The overlay join overhead on Amazon EC2

Number of nodes	Time (ms)
10	353
20	633
40	1405
80	3051
100	3604

1.5.2 Application Runtime

All tasks generated by the master are inserted into the Comet space and each should be described by XML tags which are described differently for the purpose of an application. Data to be computed can be included in a task or outside of the task such as in a file server. To show each case, let VaR tasks include data inside the tuple and Image registration tasks include data outside of the tuple because image data is relatively larger than VaR data. A typical *out* task for VaR is described as shown below.

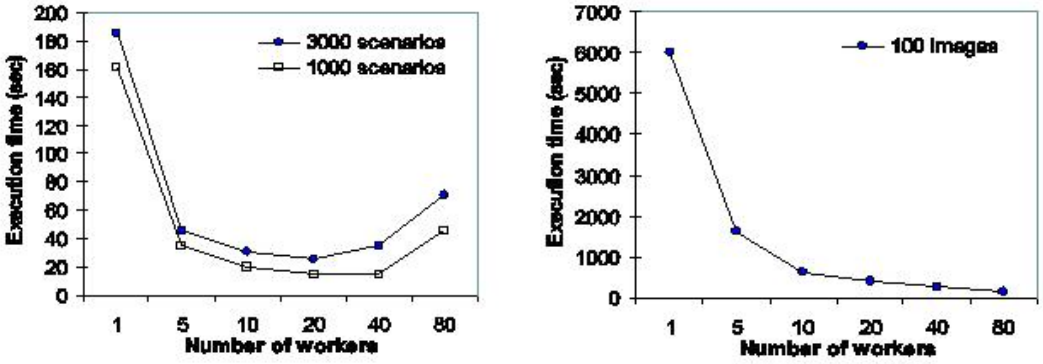
```
<VarAppTask>
   <TaskId>taskid</TaskId>
   <DataBlock>data_blocks</DataBlock>
   <MasterNetName>master_name</MasterNetName>
</VarAppTask>
```

In image registration, each worker processes a whole image, hence the number of images to be processed is the number of tasks. Besides, because the image size is too large to be conveyed on a task, when the master generates tasks, it just includes the data location for the task as a tag. After a worker takes a task from the Comet space, it connects to the data location and gets data. A typical *out* task for image registration is described as shown below.

```
<ImageRegAppTask>
   <TaskId>taskid</TaskId>
   <ImageLocation>image_location</ImageLocation>
   <MasterNetName>master_name</MasterNetName>
</ImageRegAppTask>
```

Figure 1.8 shows the total application runtime of CometCloud-based (a) VaR and (b) image registration on Amazon EC2 for different number of scenarios. In this experiment, we ran a master on the Rutgers cloud and up to 80 workers on EC2 instances. Each worker ran on a different instance. We assumed that all workers were unsecured and did not share the Comet space. As shown in (a), and as expected, the application runtime of VaR decreases as the number of EC2 workers increases up to some points. However, when the number of workers is larger than some values, the application runtime increases (see 40 and 80 workers). This is because of the communication overhead that workers ask tasks to the proxy. Note that the proxy is the access point for unsecured workers even though a request handler sends a task to the worker after the proxy forwards the request to the request handler. If the computed data size is large and it needs more time to be completed, then workers will less access the proxy and the communication overhead of the proxy will decrease. Figure 1.8 (b) shows the performance improvement of image registration when the number of workers increases. As the same as in VaR, when the number of workers increases, the application runtime

decreases. In this application, one image takes time around 1 minute to be completed, hence the communication overhead does not appear in the graph.



(a) VaR (b) Image Registration

Figure 1.8: Evaluation of CometCloud-based applications on Amazon EC2.

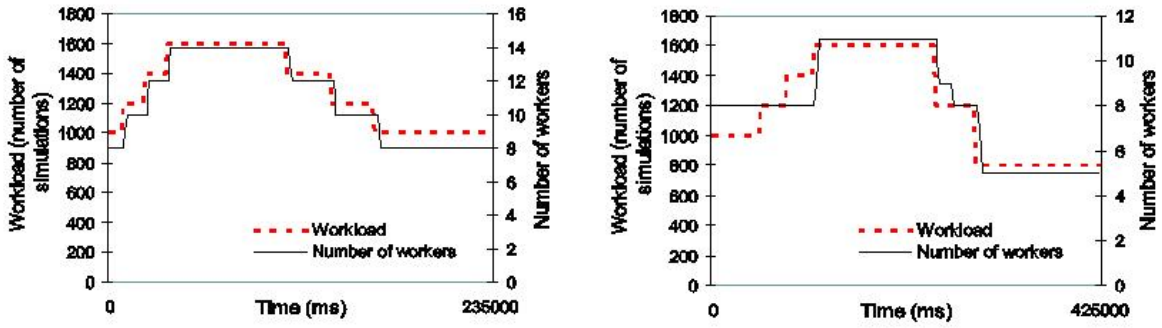
1.5.3 Autonomic Cloudbursts Behaviors

VaR using workload-based policy

In this experiment, autonomic cloudburst is represented by the number of changing workers. When the application workload increases (or decreases), a pre-defined number of workers are added (or released), based on the application workload. Specifically, we defined *workload-specific* and *workload-bounded* policies. In workload-specific, a user can specify the workload that nodes are allocated or released. In workload-bounded, whenever the workload increases by more than a specified threshold, a predefined number of workers is added. Similarly, if the workload decreases by more than the specified threshold, the predefined number of workers is released.

Figure 1.9 demonstrate autonomic cloudbursts in CometCloud based on two of the above polices, i.e., workload-specific and workload-bounded. The figure plots the changes in the number of worker as the workload changes. For the workload-specific

policy, the initial workload is set to 1000 simulations and the initial number of workers is set to 8. The workload is then increased or decreased workload by 200 simulations at a time and the number of worked added or released set to 3. For workload-bounded policy, the number of workers is initially 8 and the workload is 1000 simulations. In this experiment, the workload is increased by 200 and decreased by 400 simulations, and 3 workers are added or released at a time. The plots in Figure 1.9 clearly demonstrate the cloudburst behavior. Note that the policy used as well as the thresholds can be changed on-the-fly.



(a) Workload-specific policy (b) Workload-bounded policy

Figure 1.9: Policy-based autonomic cloudburst using CometCloud.

Image registration using budget-based policy

The virtual cloud environment used for the experiments consisted of two research sites located at Rutgers University and University of Medicine and Dentistry of New Jersey, one public cloud, i.e., Amazon Web Service (AWS) EC2 [1], and one private datacenter at Rutgers, i.e., TW. The two research sites hosted their own image servers and job queues, and workers running on EC2 or TW access these image servers to get the image described in the task assigned to them (see Figure 1.5). Each image server has 250 images resulting in a total of 500 tasks. Each image is two dimensional and its size is

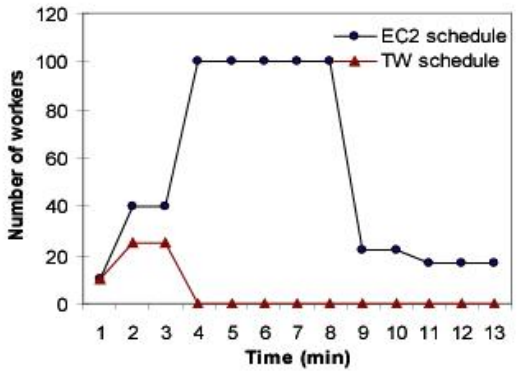
Costs for the TW datacenter included hardware investment, software, electricity etc., and were estimated based on the discussion in [12], which says that a datacenter costs $120K/lifecycle per rack and has a lifecycle of 10 years. Hence, we set the cost for TW to $1.37/hour per rack. In the experiments we set the maximum number of available nodes to 25 for TW and 100 for EC2. Note that TW nodes outperform EC2 nodes, but are more expensive. We used budget-based policy for scheduling where the scheduling agent tries to complete tasks as soon as possible without violating the budget. We set the maximum available budget in the experiments to $3 to complete all tasks. The motivation for this choice is as follows. If the available budget was sufficiently high, then all the available nodes on TW will be allocated, and tasks would be assigned until the all the tasks were completed. If the budget is too small, the scheduling agent would not be able to complete all the tasks within the budget. Hence, we set the budget to an arbitrary value in between. Finally, the monitoring component of the scheduling agent evaluated the performance every 1 minute.

Evaluation of CometCloud-based image registration application enabled scheduling agent:

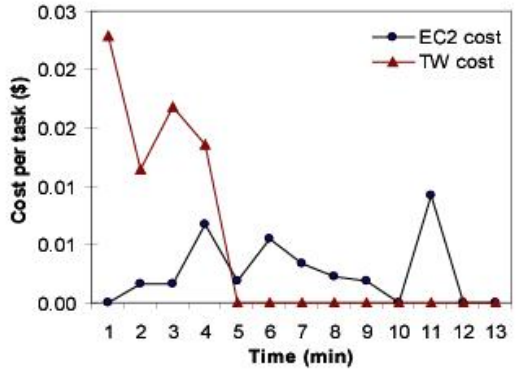
The results from the experiments are plotted in Figure 1.10. Note that since the scheduling interval is 1 min, the X-axis corresponds to both time (in minutes) and the scheduling iteration number. Initially, the CometCloud scheduling agent does not know the cost of completing a task. Hence, it initially allocated 10 nodes each from TW and EC2. Figure 1.10 (a) shows the scheduled number of workers on TW and EC2 and (b) shows costs per task for TW and EC2. In the beginning, since the budget is sufficient, the scheduling agent tries to allocate TW nodes even though they cost more than EC2 node. In the 2nd scheduling iteration, there are 460 tasks still remaining, and the agent attempts to allocate 180 TW nodes and 280 EC2 nodes to finish all tasks as soon as possible within the available budget. If TW and EC2 could provide the requested nodes, all the tasks would be completed by next iteration. However, since the maximum available number of TW nodes is only 25, it allocates these 25 TW nodes and estimates that a completion time of 7.2 iterations. The agent then decides on the number of EC2 workers to be used based on the estimated rounds.

In case of the EC2, it takes around 1 minutes to launch (from the start of virtual machine to ready state for consuming tasks), and as a results, by the 4th iteration the cost per task for EC2 increases. At this point, the scheduling agent decides to decrease the number of TW nodes, what are expensive, and instead, decides to increase the number of EC2 nodes using the available budget. By the 9th iteration, 22 tasks are still remaining. The scheduling agent now decides to release 78 EC2 nodes because they will not have jobs to execute. The reason why the remaining jobs have not completed at the 10th iteration (i.e., 10 minutes) even though 22 nodes are still working is that the performance

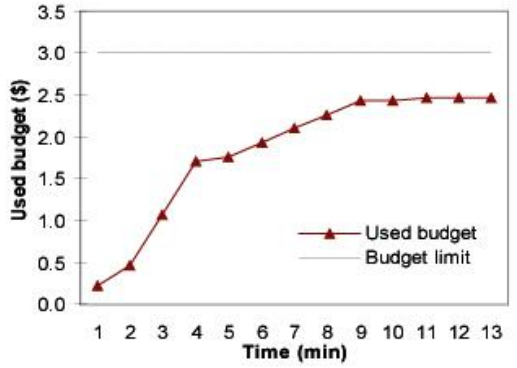
of EC2 decreased for some reason in our experiments. Figure 1.10 (c) shows the used budget over time. It shows all the tasks were completed within the budget and took around 13 minutes.



(a) Scheduled number of nodes.



(b) Calculated cost per task.

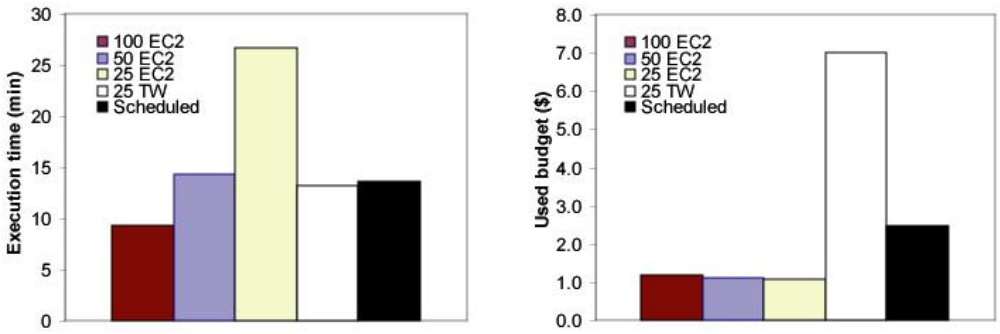


(c) Cumulative budget usage over time.

Figure 1.10: Experimental evaluation of medical image registration using CometCloud - Results using the scheduling agent.

Comparison of execution time and used budget with/without scheduling agent:

Figure 1.11 shows a comparison of execution time and used budget with/without the CometCloud scheduling agent. In the case where only EC2 nodes are used, when the number of EC2 nodes is decreased from 100 to 50 and 25, the execution time increases and the used budget decreases as shown in (a) and (b). Comparing the same number of EC2 and TW nodes (25 EC2 and 25 TW), the execution time for 25 TW nodes is approximately half that for 25 EC2 nodes, however the costs for 25 TW nodes is significantly more than that for 25 EC2 nodes. When the CometCloud autonomic scheduling agent is used, the execution time is close to that obtained using 25 TW nodes, but the cost is much smaller and the tasks are completed within the budget. An interesting observation from the plots is that if you don't have any limits on the number of EC2 nodes used, then a better solution is to allocate as many EC2 nodes as you can. However, if you only have limited number of nodes to use and want to be guaranteed that your job is completed in limited budget, then the autonomic scheduling approach achieves and an acceptable tradeoff. Note that launching EC2 nodes at runtime impacts application performance since it takes about a minute - a node launched at time *t* minutes only starts working at time *t+1* minutes. Since different cloud service will have different performance and cost profiles, the scheduling agent will have to use historical data and more complex models to compute schedules, as we extend CometCloud to include other service providers.



(a) Execution time varying the number of nodes for EC2 and TW

(b) Used budget over time varying the number of nodes for EC2 and TW

Figure 1.11: Experimental evaluation of medical image registration using CometCloud - Comparison of performance and costs with/without autonomic scheduling.

1.6. Conclusion and Future Research Directions

In this chapter, we investigated autonomic cloudbursting and autonomic cloudbridging to support real-world applications such as VaR and a medical image registration application using the CometCloud framework. CometCloud enables a virtual computational cloud that integrates local computational environments (datacenters) and public cloud services on-the-fly and provides a scheduling agent to manage cloudbridging. CometCloud supports online risk analytics which should be time-critically completed and has dynamically changing workload and medical informatics which has large data and receives requests from different distributed researcher groups with varied computational requirements and QoS constraints. The policy-driven scheduling agent uses the QoS constraints along with performance history and the state of the resources to determine the appropriate size and mix of the public and private cloud resources that should be allocated to a specific request. These applications were deployed on private clouds at Rutgers University, the Cancer Institute of New Jersey and a public cloud at Amazon EC2. The results demonstrated the effectiveness of autonomic cloudbursts as well as

policy-based autonomic scheduling and showed the feasibility to run similar types of applications using CometCloud.

We are supporting high level application models such as Hadoop/MapReduce and workflow abstraction. Also, we are deploying more applications such as Ensemble Kalman Filter, Partial Differential Equation (PDE), pharmaceutical informatics, Mandelbrot, and replica exchange on Amazon EC2. To prove the feasibility of autonomic cloudbursting and autonomic cloudbridging for more heterogeneous clouds, we plan to extend the virtual cloud such as TeraGrid and Eucalyptus.

Acknowledgements

The authors would like to thank Zhen Li, Shivangi Chaudhari and Lin Yang for their contributions to this work. The research presented in this chapter was supported in part by National Science Foundation via grants numbers IIP 0758566, CCF-0833039, DMS-0835436, CNS 0426354, IIS 0430826, and CNS 0723594, by Department of Energy via the grant number DE-FG02-06ER54857, by The Extreme Scale Systems Center at ORNL and the Department of Defense, and by an IBM Faculty Award, and was conducted as part of the NSF Center for Autonomic Computing at Rutgers University. Experiments on the Amazon Elastic Compute Cloud (EC2) were supported by a grant from Amazon Web Services.

References

[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.
[2] Google app engine. http://code.google.com/appengine/.
[3] Azure service platform. http://www.microsoft.com/azure/.
[4] Gogrid. http://www.gogrid.com.

[5] Z. Li and M. Parashar, A computational infrastructure for grid-based asynchronous parallel applications, in *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2007, pp. 229–230.

[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, 2001, pp. 149–160.

[7] C. Schmidt and M. Parashar, Squid: Enabling search in dht-based systems, *J. Parallel Distrib. Comput.*, vol. 68, no. 7, pp. 962–975, 2008.

[8] C. Schmidt and M. Parashar, Enabling flexible queries with guarantees in p2p systems, *IEEE Internet Computing*, vol. 8, no. 3, pp. 19–26, 2004.

[9] N. Carriero and D. Gelernter, Linda in context, *Commun. ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[10] J. Dean and S. Ghemawat, Mapreduce: simplified data processing on large clusters, *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] Hadoop. http://hadoop.apache.org/core/.

[12] Apc. http://apcmedia.com/salestools/CMRP-5T9PQG R3 EN.pdf.

[13] A. Meucci, Risk and Asset Allocation. New York, NY: Springer-Verlag, 2005.

[14] T. Vercauteren, X. Pennec, A. Perchant, and N. Ayache, Symmetric logdomain diffeomorphic registration: A demons-based approach, *Proc. International Conference on Medical Image Computing and Computer Assisted Intervention*, vol. 5241, pp. 754–761, 2008.

[15] C. DeLorenzo, X. Papademetris, K. Wu, K. P. Vives, D. Spencer, and J. S. Duncan, Nonrigid 3D brain registration using intensity/feature information, *Proc. International Conference on Medical Image Computing and Computer Assisted Intervention*, vol. 4190, pp. 1611–3349, 2004.

[16] B. Fischer and J. Modersitzki, Combination of automatic and landmark based registration: the best of both worlds, *SPIE Medical Imaging*, pp. 1037–1047, 2003.

[17] A. Azar, C. Xu, X. Pennec, and N. Ayache, An interactive hybrid nonrigid registration framework for 3D medical images, *Proc. International Symposium on Biomedical Imaging*, pp. 824–827, 2006.

[18] L. Yang, L. Gong, H. Zhang, J. L. Nosher, and D. J. Foran, A method for parallel platform based acceleration of landmark based image registration, in *Proc. European Conference on Parallel Computing*, Netherlands, 2009.

[19] M. A. Fischler and R. C. Bolles, Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography, *Comm. of the ACM*, vol. 24, pp. 381–395, 1981.

[20] H. Chui and A. Rangarajan, A new point matching algorithm for nonrigid registration, *Computer Vision and Image Understanding*, vol. 89, no. 2, pp. 114–141, 2003.

[21] R. Tolksdorf and D. Glaubitz, Coordinating web-based systems with documents in xmlspaces, In *Proceedings of the Sixth IFCIS International Conference on Cooperative Information Systems*, pp. 356–370. Springer Verlag, 2001.

[22] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, Analysis of the clustering properties of the hilbert space-filling curve, *IEEE Transactions on Knowledge and Data Engineering*, vol. 13 no. 1 pp. 124–141, 2001.

[23] D. Nurmi, R.Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov, The Eucalyptus Open-source Cloudcomputing System, *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 124-131, 2009.