Autonomic Metered Pricing for a Utility Computing Service

Chee Shin Yeo, Srikumar Venugopal, Xingchen Chu, Rajkumar Buyya *

Grid Computing and Distributed Systems Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, VIC 3010, Australia

Abstract

An increasing number of providers are offering utility computing services which require users to pay only when they use. Most of these providers currently charge users for metered usage based on fixed prices. In this paper, we analyze the pros and cons of charging fixed prices as compared to variable prices. In particular, charging fixed prices do not differentiate pricing based on different user requirements. Hence, we highlight the importance of deploying an autonomic pricing mechanism that self-adjusts pricing parameters to consider both application and service requirements of users. Performance results observed in the actual implementation of an enterprise Cloud show that the autonomic pricing mechanism is able to achieve higher revenue than various other common fixed and variable pricing mechanisms.

Key words: Service Pricing, Autonomic Management, Advanced Reservation, Quality of Service (QoS), Utility Computing, Cloud Computing

1. Introduction

The next era of computing is envisioned to be that of *utility computing* [1]. The vision of utility computing is to provide computing services to users on demand and charge them based on their usage and Quality of Service (QoS) expectations. Users no longer have to invest heavily in or maintain their own computing infrastructure. Instead, they employ computing services offered by providers to execute their applications. This commoditized computing model thus strengthens the case to charge users via metered usage [2], just like in real-world utilities. In other words, users only have to pay for what they use.

The latest emergence of Cloud computing [3] is a significant step towards realizing this utility computing model since it is heavily driven by industry vendors. Cloud computing promises to deliver reliable services through next-generation data centers built on virtualized compute and storage technologies. Users will be able to access applications and data from a "Cloud" anywhere in the world on demand and pay based on what they use. As more providers are starting to offer pay-per-use utility computing services using Cloud infrastructure, the issue of how to determine the right price for users is now becoming increasingly critical for these providers. This is because pricing is able to regulate the supply and demand of computing services and thus affects both providers (who supply the services) and users (who demand the services) respectively. When the right price is set, a provider can not only attract/restrict a sufficient number of users to meet its revenue target, but also provide computing services more effectively and efficiently to meet the

* Corresponding author.
Email addresses: csyeo@csse.unimelb.edu.au (Chee Shin Yeo), srikumar@csse.unimelb.edu.au (Srikumar Venugopal), xchu@csse.unimelb.edu.au (Xingchen Chu), raj@csse.unimelb.edu.au (Rajkumar Buyya).

service needs of users. Hence, the aim of this paper is to justify the need for an autonomic pricing mechanism that can set this right price for a provider. In particular, this paper focuses on how a provider can charge commercial users or enterprises which make heavy demands on computing resources, as compared to personal users or individuals with considerably lower requirements.

Currently, providers follow a fairly simple pricing scheme to charge users – *fixed* prices based on various resource types. For processing power (as of 15 October 2008), Amazon [4] charges $0.10 per virtual computer instance per hour (Hr), Sun Microsystems [5] charges $1.00 per processor (CPU) per Hr, and Tsunamic Technologies [6] charges $0.77 per CPU per Hr. Instead of charging fixed prices for these heavy users, we advocate charging *variable* prices and provide guaranteed QoS through the use of advanced reservations. Advance reservations are bookings made in advance to secure an available item in the future and are used in the airline, car rental, and hotel industries. In the context of utility computing, an advance reservation is a guarantee of access to a computing resource at a particular time in the future for a particular duration [7].

Charging fixed prices in utility computing is not fair to both the provider and users since different users have distinctive needs and demand specific QoS for various resource requests that can change anytime. In economics, a seller with constrained capacity can adjust prices to maximize revenue if the following four conditions are satisfied [8]: (i) demand is variable but follows a predictable pattern, (ii) capacity is fixed, (iii) inventory is perishable (wasted if unused), and (iv) seller has the ability to adjust prices. Thus, for utility computing, providers can charge variable prices since: (i) demand for computing resources changes but can be expected using advanced reservations [7], (ii) only a limited amount of resources is available at a particular site owned by a provider, (iii) processing power is wasted if unused, and (iv) a provider can change prices.

The main aim of providers charging variable prices is to maximize revenue by differentiating the value of computing services provided to different users. Since providers are commercial businesses driven by profit, they need to maximize revenue. Profitable providers can then fund further expansions and enhancements to improve their utility computing service offerings. Charging variable prices is also particularly useful for resource management as it can result in the diversion of demand from high-demand time periods to low-demand time periods [8], thus maximizing utilization for a utility computing service. Higher prices increase revenue as users who need services during high-demand time periods are willing to pay more, whereas others will shift to using services during low-demand periods. The latter results in higher utilization during these otherwise underutilized low-demand periods and hence leads to higher revenue.

In general, fixed prices are simpler to understand and more straightforward for users as compared to variable prices. However, all users do not have the same need. Hence, it is not fair for all users to be charged the same fixed price since not all users may afford the same price. Fixed prices also do not allow price-sensitive users to benefit from lower prices which they prefer to accept in exchange of certain restrictions. Moreover, fixed prices do not permit a provider to give specific incentives via differentiated pricing based on distinct user requirements, which is the emphasis of this paper. In this paper, we propose charging variable prices with advanced reservations so that users are not only able to secure their required resources in advance, but also know the exact expenses which is computed during the time of reservation (even though they are based on variable prices). This will continue to enable users to perform budgeting with known variable prices in advance as in the case of fixed prices.

This paper proposes an autonomic pricing mechanism for a utility computing service which automatically adjusts prices when necessary to increase revenue. In particular, we highlight the significance of considering essential user requirements that encompass application and service requirements. Pricing computing resources according to user requirements benefits the utility computing service since different users require specific needs to be met and are willing to pay varying prices to achieve them. The key contributions of this paper are to:

- Consider two essential user requirements for autonomic metered pricing: (i) application and (ii) service.
- Describe how metered pricing can be implemented in an enterprise Cloud with advanced reservations.
- Analyze the performance of various fixed and variable pricing mechanisms through experimental results to demonstrate the importance of autonomic metered pricing.

This paper is organized as follows: Section 2 discusses related work. Section 3 examines economic

aspects of a utility computing service. Section 4 describes the implementation of metered pricing using a service-oriented enterprise Cloud technology called Aneka [9]. Section 5 explains the evaluation methodology and experimental setup to assess the performance of various fixed and variable pricing mechanisms with respect to the application and service requirements of users. Section 6 analyzes the performance results. Section 7 presents conclusions and future work.

2. Related Work

Many market-based resource management systems have been implemented across numerous computing platforms [10] including clusters, distributed databases, Grids, parallel and distributed systems, peer-to-peer, and the World Wide Web. To manage resources, these systems adopt a variety of economic models [11], such as auction, bargaining, bartering, commodity market, bid-based proportional resource sharing, posted price, and tendering/contract-net. In this paper, we examine metered pricing which is applicable in commodity market and posted price models.

Recently, several works have discussed pricing for utility or on-demand computing services. In particular, these works have identified and addressed various distinguished features of utility computing which is different from traditional pricing mechanisms in economics. Price-At-Risk [12] considers uncertainty in the pricing decision for utility computing services which have uncertain demand, high development costs, and short life cycle. Pricing models for on-demand computing [13] have been proposed based on various aspects of corporate computing infrastructure which include cost of maintaining infrastructure in-house, business value of infrastructure, scale of infrastructure, and variable costs of maintenance. Another work [14] considers economic aspects of a utility computing service whereby high prices denote higher service level for faster computation. But, these works do not consider autonomic pricing that addresses users' application requirements (such as parallel applications) and service requirements (such as deadline and budget).

Setting variable prices is known as *price discrimination* in economics [15]. Sulistio et al. [16] have examined third degree price discrimination by using revenue management [8] to determine the pricing of advanced reservations in Grids. It evaluates revenue performance across multiple Grids for variable pricing based on the combination of three market segments of users (premium, business, and budget) and three time periods of resource usage (peak, off-peak, and saver). Hence, it does not derive fine-grained variable prices that differentiate specific application and service requirements of individual users.

Chen et al. [17] have proposed pricing-based strategies for autonomic control of web servers. It uses pricing and admission control mechanisms to control QoS of web requests such as slowdown and fairness. However, this paper focuses on high-performance applications and user-centric service requirements (deadline and budget).

In our previous work, we have presented Libra [18] as a market-based solution for delivering more utility to users in clusters compared to traditional scheduling policies. As Libra only computes a static cost, an extension called Libra+$ [19] uses an enhanced pricing function that satisfies four essential requirements for pricing of resources to prevent workload overload: (i) flexible, (ii) fair, (iii) dynamic, and (iv) adaptive. In this paper, we propose an autonomic version of Libra+$ called Libra+$Auto which is feasible as demonstrated through its actual implementation in an enterprise Cloud.

Libra+$Auto has a number of pros compared to Libra+$ and Libra. First, Libra+$Auto is able to automatically adjust pricing parameters and hence does not rely on static pricing parameters to be configured manually by the provider in the case of both Libra+$ and Libra. Second, Libra+$Auto considers the current workload across nodes when computing prices, whereas Libra+$ only consider the current workload within a node and Libra does not consider any current workload at all. Third, Libra+$Auto can exploit the budget limits of users to improve the revenue of the provider by automatically adjusting to increase prices when there are fewer available compute nodes and reduce prices when there are more available nodes. Fourth, Libra+$Auto offers more precise incentives to individual users which can promote user demand and in turn improve revenue, since it dynamically changes prices in a more fine-grained manner than both Libra+$ and Libra via expected workload demand and availability of nodes. On the other hand, Libra+$Auto has only one con compared to Libra+$ and Libra, which is requiring more computation time to determine the availability of nodes and adjust prices depending on the acceptance of previous requests.

3. Economic Aspects of a Utility Computing Service

This section examines various economic aspects of a utility computing service including: (i) variable pricing with advanced reservations, (ii) pricing issues, and (iii) pricing mechanisms. We consider a scenario wherein a provider owns a set of resources that we term as compute nodes. Each node can be subdivided into resource partitions and leased to users for certain time intervals.

3.1. *Variable Pricing with Advanced Reservations*

The use of advanced reservations has been proposed to provide QoS guarantees for accessing various resources across independently administered systems such as Grids [7]. With advanced reservations, users are able to secure resources required in the future which is important to ensure successful completion of time-critical applications such as real-time and workflow applications or parallel applications requiring a number of processors to run. The provider is able to predict future demand and usage more accurately. Using this knowledge, the provider can apply revenue management [8] to determine pricing at various times to maximize revenue. Once these prices are advertised, users are able to decide in advance where to book resources according to their requirements and their resulting expenses.

Having prior knowledge of expected costs is highly critical for enterprises to successfully plan and manage their operations. Resource supply guarantee also allows enterprises to contemplate and target future expansion more confidently and accurately. Enterprises are thus able to scale their reservations accordingly based on short-term, medium-term, and long-term commitments.

Users may face the difficulty of choosing the best price for reserving resources from different utility computing services at different times. This difficulty can be overcome by using resource brokers [20] which act on the behalf of users to identify suitable utility computing services and compare their prices.

3.2. *Pricing Issues*

For simplicity, we examine metered pricing within a utility computing service with constrained capacity and do not consider external influences that can be controlled by the provider, such as cooperating with other providers to increase the supply of resources [21] or competing with them to increase market share [22]. Price protection and taxation regulations from authorities and inflation are beyond the control of the provider. We assume that users have to pay in order to guarantee reservations. Thus, a utility computing service requires payment from users either at the time of reservation or later depending on payment agreements so as to reserve computing resources in advance.

Table 1
Pricing for processing power.

Name	**Configured Pricing Parameters**
FixedMax	\$3/CPU/Hr
FixedMin	\$1/CPU/Hr
FixedTimeMax	\$1/CPU/Hr (12AM–12PM) \$3/CPU/Hr (12PM–12AM)
FixedTimeMin	\$1/CPU/Hr (12AM–12PM) \$2/CPU/Hr (12PM–12AM)
Libra+\$Max	\$1/CPU/Hr ($PBase_j$), $\alpha = 1$, $\beta = 3$
Libra+\$Min	\$1/CPU/Hr ($PBase_j$), $\alpha = 1$, $\beta = 1$
Libra+\$Auto	same as Libra+\$Min

We also assume that the execution time period of applications will be within the reservation time period. In order to enforce other scheduled reservations, a utility computing service will terminate any outstanding applications that are still executing once the time period of reservation expires. This implies that users must ensure that time periods of reservations are sufficient for their applications to be completed. Therefore, users may have to reserve more time to protect their applications from forced termination if they are uncertain whether their applications will take more time to execute than estimated. Although this restriction is unfavorable for users, users can try to minimize its impact by using runtime prediction models [23][24] to estimate their application runtimes more accurately. Users are also personally responsible for ensuring that their applications can fully utilize the reserved resources. It is thus disadvantageous to the users if their applications fail to use the entire amount of reserved resources that they have already paid for.

3.3. *Pricing Mechanisms*

We compare three types of pricing mechanisms: (i) Fixed, (ii) FixedTime, and (iii) Libra+$. As listed

in Table 1, each pricing mechanism has maximum and minimum types which are configured accordingly to highlight the performance range of the pricing mechanism. Fixed charges a fixed price for per unit of resource partition at all times. FixedTime charges a fixed price for per unit of resource partition at different time periods of resource usage where a lower price is charged for off-peak (12AM–12PM) and a higher price for peak (12PM–12AM).

Libra+$ [19] computes the price P_{ij} for per unit of resource partition utilized by reservation request i at compute node j as: $P_{ij} = (\alpha * PBase_j) + (\beta * PUtil_{ij})$. The base price $PBase_j$ is a static pricing component for utilizing a resource partition at node j which can be used by the provider to charge the minimum price so as to recover the operational cost. The utilization price $PUtil_{ij}$ is a dynamic pricing component which is computed as a factor of $PBase_j$ based on the availability of the resource partition at node j for the required deadline of request i: $PUtil_{ij} = RESMax_j/RESFree_{ij} * PBase_j$. $RESMax_j$ and $RESFree_{ij}$ are the maximum units and remaining free units of the resource partition at node j for the deadline duration of request i respectively. Thus, $RESFree_{ij}$ has been deducted units of resource partition committed for other confirmed reservations and request i for its deadline duration.

The factors α and β for the static and dynamic components of Libra+$ respectively, provide the flexibility for the provider to easily configure and modify the weight of the static and dynamic components on the overall price P_{ij}. Libra+$ is fair since requests are priced based on the amount of different resources utilized. It is also dynamic because the overall price of a request varies depending on the availability of resources for the required deadline. Finally, it is adaptive as the overall price is adjusted (depending on the current supply and demand of resources) to either encourage or discourage request submission.

Fixed, FixedTime, and Libra+$ rely on static pricing parameters that are difficult to be accurately derived by the provider to produce the best performance where necessary. Hence, we propose Libra+$Auto, an autonomic Libra+$ that automatically adjusts β based on the availability of compute nodes. Libra+$Auto thus considers the pricing of resources across nodes, unlike Libra+$ which only considers pricing of resources at each node j via P_{ij}.

Algorithm 1 shows the pseudocode for adjusting β in Libra+$Auto. First, the previous dynamic factor

Algorithm 1: Pseudocode for adjusting β in Libra+$Auto.

```
1  βPrev ← (Σ_{i=1}^{n_prev} β_i for previous request) / n ;
2  maxNodes ← maximum number of nodes ;
3  foreach node i allocated to new request do
4      freeNodes ← free number of nodes for
       proposed time slot at this node i;
5      reservedNodes ← maxNodes − freeNodes ;
6      if freeNodes = 0 then
7          freeNodes ← 1 ;
8      endif
9      ratioFree ← maxNodes / freeNodes ;
10     if reservedNodes = 0 then
11         reservedNodes ← 1 ;
12     endif
13     ratioReserved ← reservedNodes / maxNodes ;
14     if previous request meets budget then
15         β_i ← βPrev * ratioFree ;
16     else
17         β_i ← βPrev * ratioReserved ;
18     endif
19 endfch
```

$\beta Prev$ is computed as the average of dynamic factors β at n_{prev} number of allocated compute nodes for the previous reservation request (line 1). Initially, when adjusting β for the first request, $\beta Prev$ uses a default value that is given by the provider. The maximum number of nodes is also assigned (line 2). Then, the free and reserved number of nodes is determined for the proposed time slots at various nodes to be allocated for the new reservation request (line 3–5). After that, the new dynamic factor β_i for the node i is updated depending on the outcome of the previous request. $\beta Prev$ is increased to accumulate more revenue if the previous request meets the user-defined budget, otherwise it is reduced (line 14–18). For increasing $\beta Prev$, a larger increase is computed when there are less free nodes left for the proposed time slot so as to maximize revenue with decreasing capacity (line 6–9). Conversely, for reducing $\beta Prev$, a larger reduction is computed when there are more free nodes left for the proposed time slot in order not to waste unused capacity (line 10–13).

Assuming that the previous reservation request wants to reserve n_{prev} number of nodes, it takes $O(n_{prev})$ time to compute the previous dynamic factor $\beta Prev$ (line 1). In addition, it takes $O(n_{new})$ time to compute the new dynamic factor β_i at each node i for n_{new} number of nodes required by the new reservation request (line 3–19). When there is maximum m number of nodes that can be possibly allocated and searching through the entire data struc-

ture containing all reserved time slots takes $O(ts)$ time in the worst case (depending on the type of data structure which is used), it takes $O((m-1).O(ts))$ time to determine the free number of nodes for the proposed time slot at node i (line 4). Hence, adjusting β in Libra+$Auto can take $O(n_{prev}+n_{new}.(m-1).O(ts))$ time in the worst case.

4. System Implementation

This section describes how metered pricing for a utility computing service can be implemented using a .NET-based service-oriented enterprise Cloud technology called Aneka [9]. Our implementation in Aneka uses advanced reservations to guarantee dedicated access to computing resources for required time periods in the future.

4.1. *Aneka: Enterprise Cloud Technology*

Aneka [9] is designed to support multiple application models, persistence and security solutions, and communication protocols such that the preferred selection can be changed at anytime without affecting an existing enterprise Cloud. To create an enterprise Cloud, the provider only needs to start an instance of the configurable Aneka container hosting required services on each enterprise Cloud node.

The purpose of the Aneka container is to initialize services and acts as a single point for interaction with the entire enterprise Cloud. To support scalability, the Aneka container is designed to be lightweight by providing the bare minimum functionality needed for an enterprise Cloud node. It provides the base infrastructure that consists of services for persistence, security (authorization, authentication and auditing), and communication (message handling and dispatching).

The Aneka container can host any number of optional services that can be added to augment the capabilities of an enterprise Cloud node. Examples of optional services are information and indexing, scheduling, execution, and storage services. This provides a flexible and extensible framework or interface for the provider to easily support various application models, including MapReduce [25] which is often associated with Cloud computing systems. Thus, resource users can seamlessly execute different types of application in an enterprise Cloud.

To support reliability and flexibility, services are designed to be independent of each other in a Aneka container. A service can only interact with other services on the local node or other nodes through known interfaces. This means that a malfunctioning service will not affect other working services and/or the Aneka container. Therefore, the provider can easily configure and manage existing services or introduce new ones into a Aneka container.

4.2. *Resource Management Architecture*

We implement a bi-hierarchical advance reservation mechanism for the enterprise Cloud with a Reservation Service at a master node that coordinates multiple execution nodes and an Allocation Service at each execution node that keeps track of the reservations at that node. This architecture was previously introduced by Venugopal et al. [26]. Figure 1 shows the interaction between the user/broker, the master node and execution nodes in the enterprise Cloud. To use the enterprise Cloud, the resource user (or a broker acting on its behalf) has to first make advanced reservations for resources required at a designated time in the future.

During the request reservation phase, the user/broker submits reservation requests through the Reservation Service at the master node. The Reservation Service discovers available execution nodes in the enterprise Cloud by interacting with the Allocation Service on them. The Allocation Service at each execution node keeps track of all reservations that have been confirmed for the node and can thus check whether a new request can be satisfied or not.

By allocating reservations at each execution node instead of at the master node, computation overheads arising from making allocation decisions are distributed across multiple nodes and thus minimized, as compared to overhead accumulation at a single master node. The Reservation Service then selects the required number of execution nodes and informs their Allocation Services to temporarily lock the reserved time slots. After all the required reservations on the execution nodes have been temporarily locked, the Reservation Service returns the reservation outcome and its price (if successful) to the user/broker.

The user/broker may confirm or reject the reservations during the confirm reservation phase. The Reservation Service then notifies the Allocation Service of selected execution nodes to lock or remove temporarily locked time slots accordingly. We

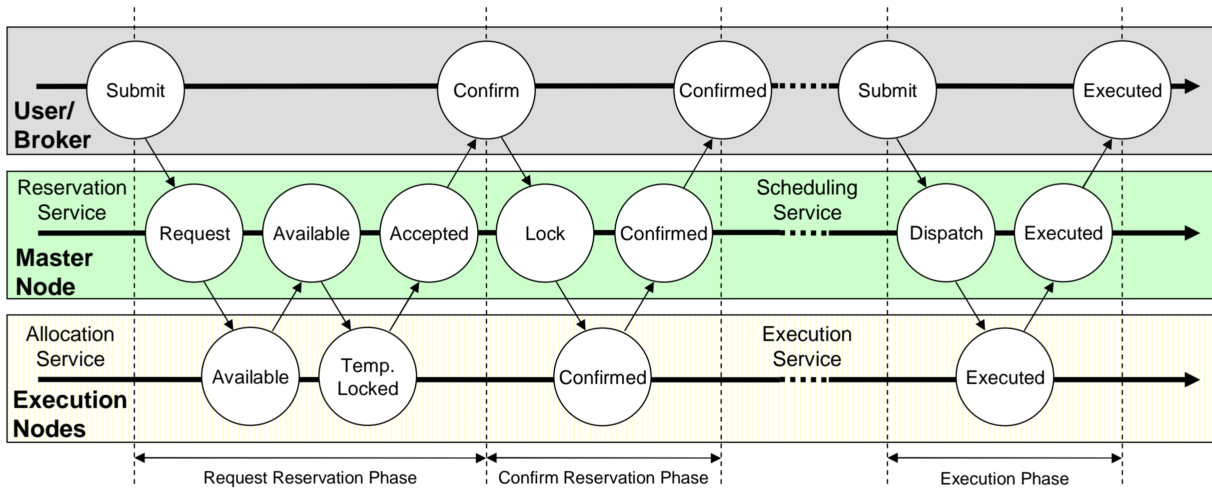


Fig. 1. Sequence of events between enterprise Cloud nodes for a successful reservation request.

assume that a payment service is in place to ensure the user/broker has sufficient funds and can successfully deduct the required payment before the Reservation Service proceeds with the final confirmation.

During the execution phase when the reserved time arrives, the user/broker submits applications to be executed to the Scheduling Service at the master node. The Scheduling Service determines whether any of the reserved execution nodes are available before dispatching applications to them for execution, otherwise applications are queued to wait for the next available execution nodes that are part of the reservation. The Execution Service at each execution node starts executing an application after receiving it from the Scheduling Service and updates the Scheduling Service of changes in execution status. Hence, the Scheduling Service can monitor executions for an application and notify the user/broker upon completion.

4.3. *Allocating Advanced Reservations*

Figure 2 shows that the process of allocating advanced reservations happens in two levels: the Allocation Service at each execution node and the Reservation Service at the master node. Both services are designed to support pluggable policies so that the provider has the flexibility to easily customize and replace existing policies for different levels and/or nodes without interfering with the overall resource management architecture.

The Allocation Service determines how to schedule a new reservation at the execution node. For simplicity, we implement the same time slot selection policy for the Allocation Service at every execution node. The Allocation Service allocates the requested time slot if the slot is available. Otherwise, it assigns the next available time slot after the requested start time that can meet the required duration.

The Reservation Service performs node selection by choosing the required number of available time slots from execution nodes and administers admission control by accepting or rejecting a reservation request. It also calculates the price for a confirmed reservation based on the implemented pricing policy. Various pricing policies considered in this paper are explained in Section 3.3. Available time slots are selected taking into account the application requirement of the user.

The application requirement considered here is the task parallelism to execute an application. A sequential application has a single task and thus needs a single processor to run, while a parallel application needs a required number of processors to concurrently run at the same time.

For a sequential application, the selected time slots need not have the same start and end times. Hence, available time slots with the lowest prices are selected first. If there are multiple available time slots with the same price, then those with the earliest start time available are selected first. This ensures that the cheapest requested time slot is allocated first if it is available. Selecting available time slots with the lowest prices first is fair and realistic. In reality, reservations that are confirmed earlier enjoy the privilege of cheaper prices, as compared to

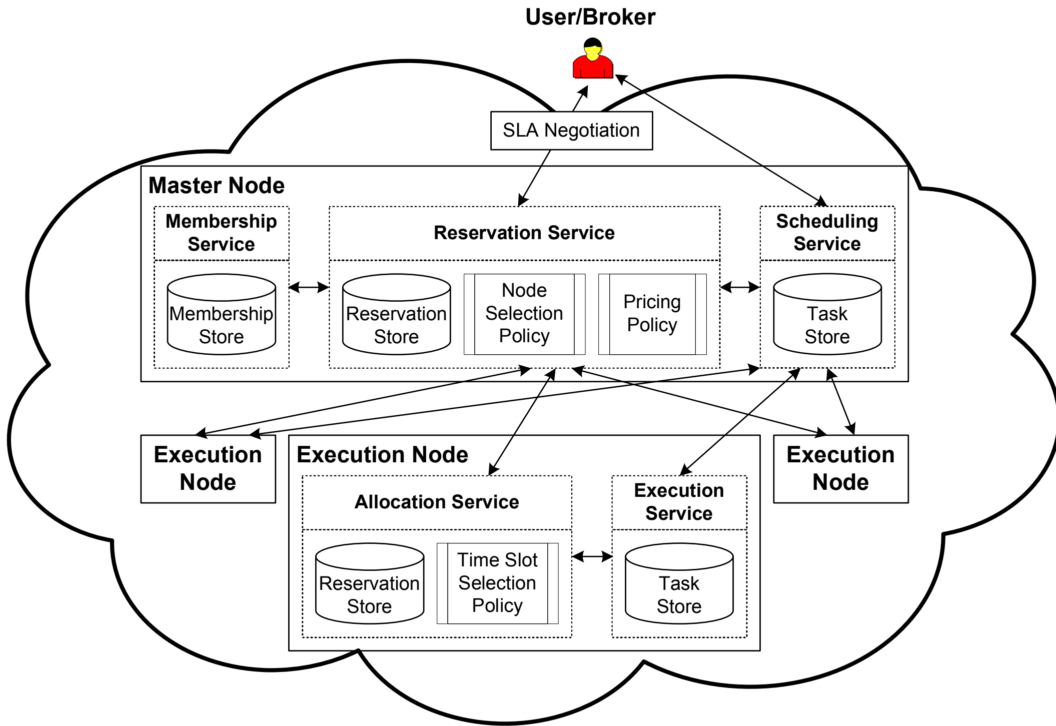


Fig. 2. Interaction of services in enterprise Cloud.

reservation requests that arrive later.

But, for a parallel application, all the selected time slots must have the same start and end times. Again, the earliest time slots (with the same start and end times) are allocated first to ensure the requested time slot is allocated first if available. If there are more available time slots (with the same start and end times) than the required number of time slots, then those with the lowest prices are selected first.

The admission control operates according to the service requirement of the user. The service requirements examined are the deadline and budget to complete an application. We currently assume both deadline and budget are hard constraints. Hence, a confirmed reservation must not end after the deadline and cost more than the budget. Therefore, a reservation request is not accepted if there is insufficient number of available time slots on execution nodes that ends within the deadline and the total price of the reservation costs more than the budget.

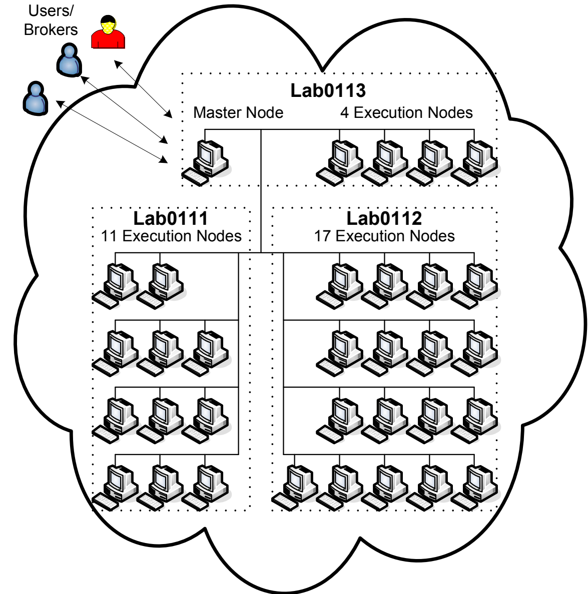


Fig. 3. Configuration of enterprise Cloud.

5. Performance Evaluation

Figure 3 shows the enterprise Cloud setup used for performance evaluation. The enterprise Cloud comprises 33 PCs providing dedicated access to computing resources through 1 master node and 32 execution nodes located across 3 student computer laboratories in the Department of Computer Science and Software Engineering, The University of Melbourne. Synthetic workloads are created by utilizing

trace data.

We use Feitelson's Parallel Workload Archive [27] to model the reservation requests because trace data of Cloud applications are currently not released and shared by any commercial Cloud service providers. But, for a scientific research paper, it is extremely important to have publicly accessible trace data so that our experiments can be reproducible by other researchers. Moreover, this paper focuses on studying the application requirements of users in the context of High Performance Computing (HPC). Hence, the Parallel Workload Archive meets our objective by providing the necessary characteristics of real parallel applications collected from supercomputing centers. Unfortunately, since the Parallel Workload Archive is not based on paying users in utility computing environments, it is possible that the trace pattern of these archived workloads will be different from those with paying users.

Our experiments utilize 238 reservation requests in the last 7 days of the SDSC SP2 trace (April 1998 to April 2000) version 2.2 from the Parallel Workload Archive. The SDSC SP2 trace from the San Diego Supercomputer Center (SDSC) in USA is chosen due to the highest resource utilization of 83.2% among available traces to ideally model a heavy workload scenario. The trace only provides the inter-arrival times of reservation requests, the number of processors to be reserved as shown in Figure 4(a) (downscaled from a maximum of 128 nodes in the trace to a maximum of 32 nodes), and the duration to be reserved as shown in Figure 4(b). Service requirements are not available in this trace. Hence, we use a methodology proposed by Irwin et al. [28] to synthetically assign service requirements through two request classes: (i) *Low Urgency (LU)* and (ii) *High Urgency (HU)*. Figure 4(b) and 4(c) show the synthetic values of deadline and budget for the 238 requests respectively.

A reservation request i in the LU class has a deadline of high $deadline_i/duration_i$ value and budget of low $budget_i/f(duration_i)$ value. $f(duration_i)$ is a function representing the minimum budget required based on $duration_i$. Conversely, each request in the HU class has a deadline of low $deadline_i/duration_i$ value and budget of high $budget_i/f(duration_i)$ value. This is realistic since a user who submits a more urgent request to be met within a shorter deadline offers a higher budget for the short notice. Values are normally distributed within each of the deadline and budget parameters.

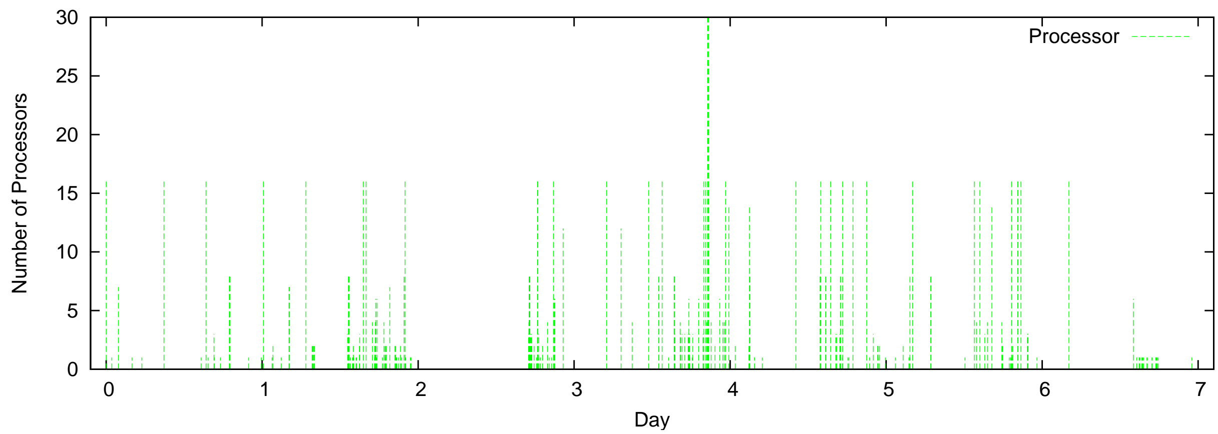
For simplicity, we only evaluate the performance of pricing for processing power as listed in Table 1 with various combinations of application requirements (sequential and parallel) and request classes (LU and HU). However, the performance evaluation can be easily extended to include other resource types such as memory, storage, and bandwidth. Both LU and HU classes are selected so as to observe the performance under extreme cases of service requirements with respective highest and lowest values for deadline and budget. We also currently assume that every user/broker can definitely accept another reservation time slot proposed by the enterprise Cloud if the requested one is not possible, provided that the proposed time slot still satisfies both application and service requirements of the user.

6. Performance Results

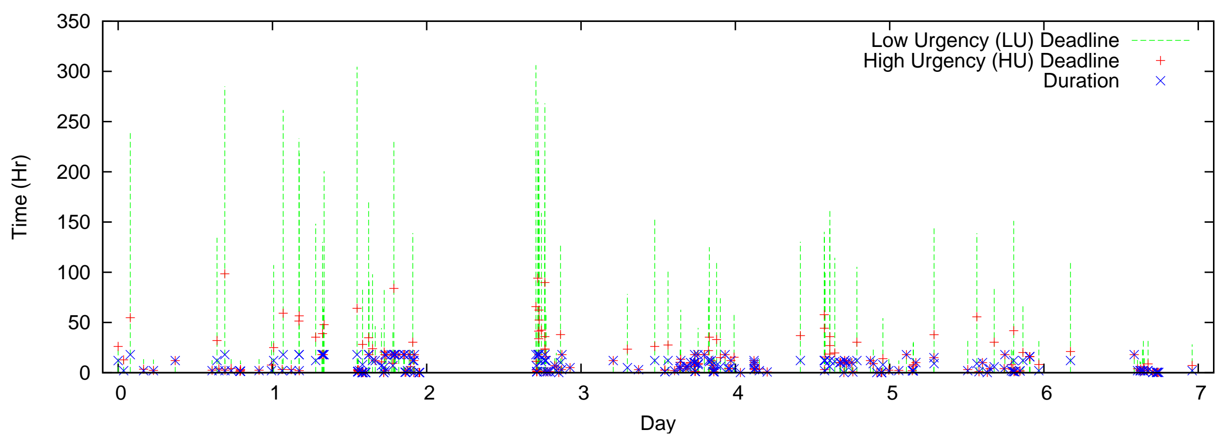
We analyze the performance results of seven various pricing mechanisms (listed in Table 1) over one week with respect to the application and service requirements of users. The three performance metrics being measured are: (i) the accumulated revenue of confirmed reservations in \$, (ii) the current average price of confirmed reservations in \$/CPU/Hr, and (iii) the accumulated number of confirmed reservations. The performance results of all three metrics have been normalized to produce standardized values within the range of 0 to 1 for easier relative comparison. The revenue (in \$) of a confirmed reservation is the total sum of revenue across all its reserved nodes calculated using the assigned price (depending on the specific pricing mechanism) and the reserved duration at each node. Then, the average price (in \$/CPU/Hr) of a confirmed reservation is computed to reflect the standard price across all its reserved nodes.

6.1. *Fixed Prices*

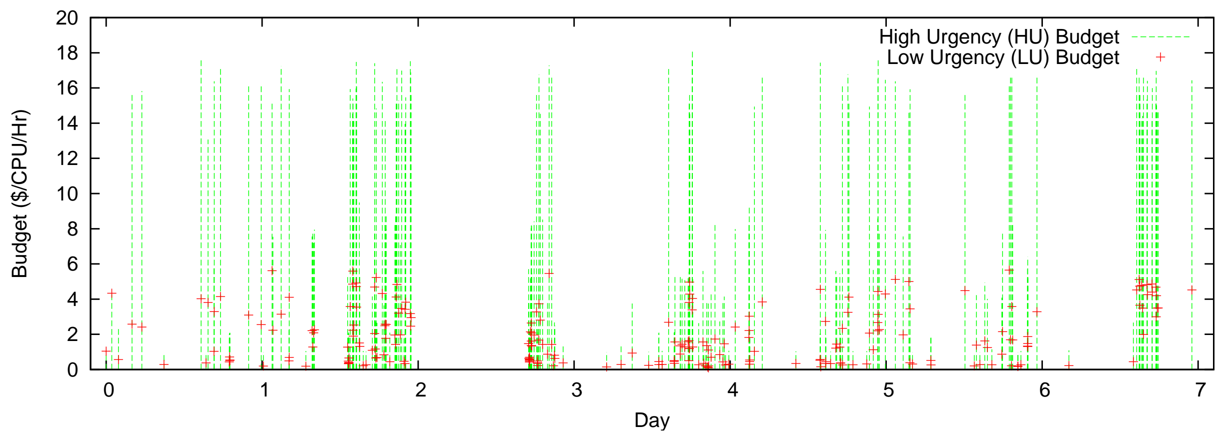
Based on the configured pricing parameters of the four fixed pricing mechanisms listed in Table 1, we can observe that FixedMax charges the highest current average price, followed by FixedTimeMax, FixedTimeMin, and FixedMin (Figure 5(b), 6(b), 7(b), and 8(b)). FixedMax acts as the maximum bound of the fixed pricing mechanisms by charging the highest price of \$3/CPU/Hr for processing power, while FixedMin acts as the minimum bound by charging the lowest price of \$1/CPU/Hr.



(a) Number of processors (from trace)

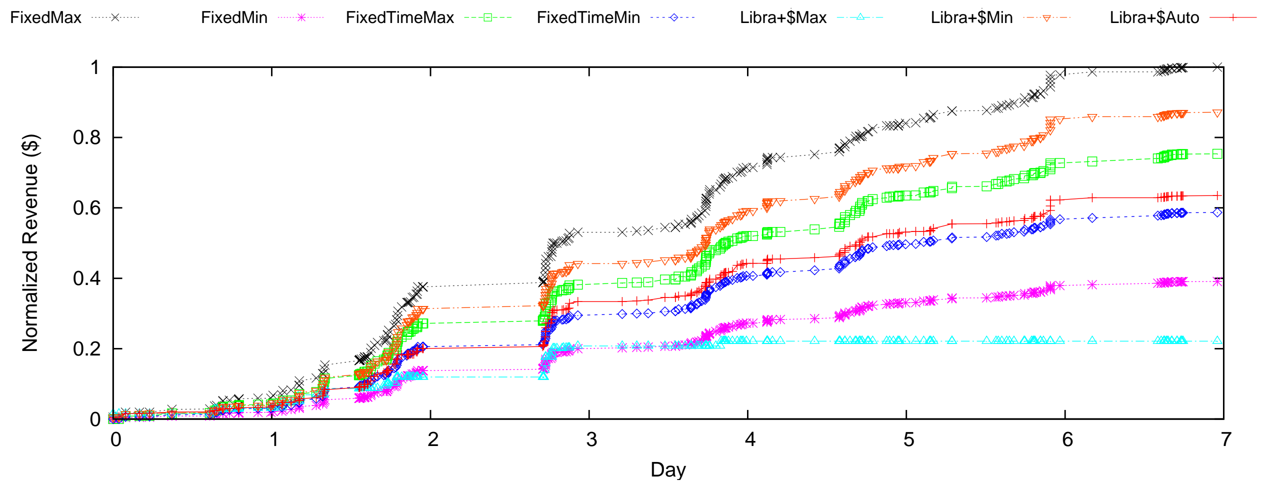


(b) Duration (from trace) and deadline (synthetic)

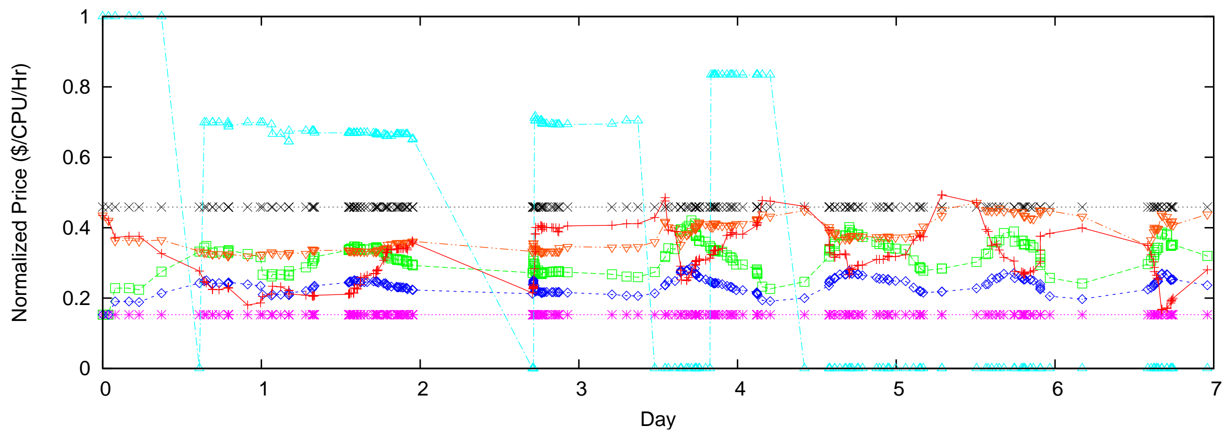


(c) Budget (synthetic)

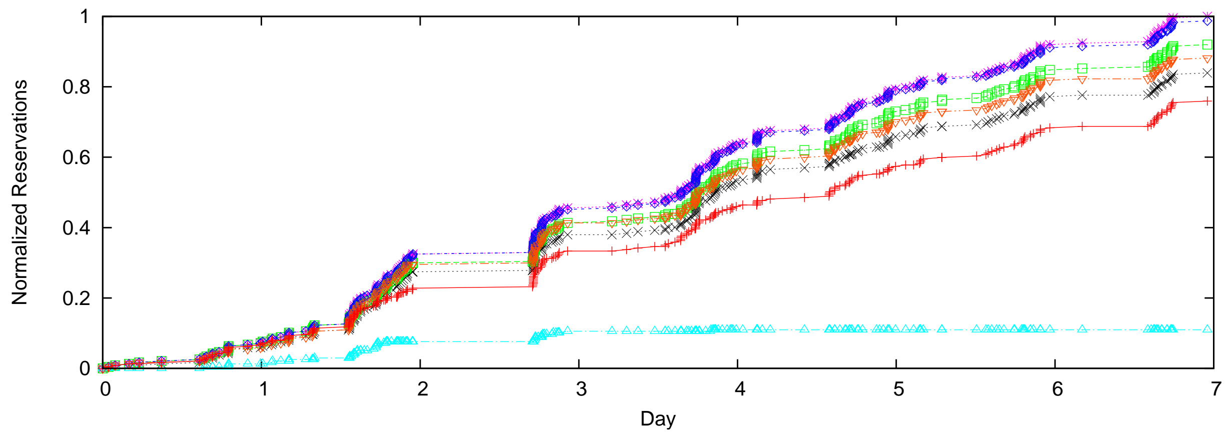
Fig. 4. Last 7 days of SDSC SP2 trace (April 1998 to April 2000) with 238 requests.



(a) Accumulated Revenue

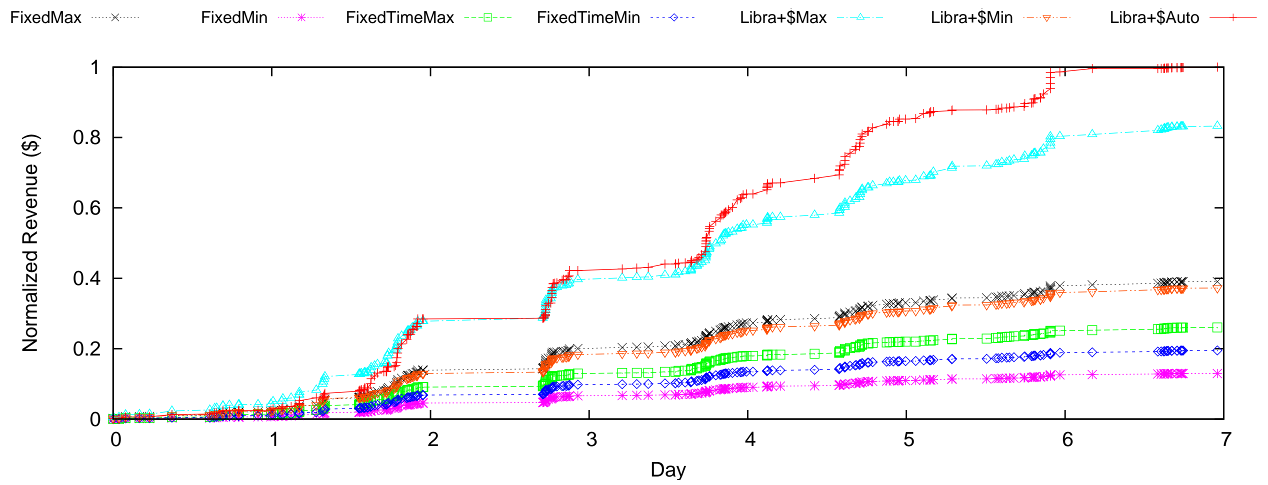


(b) Current Average Price

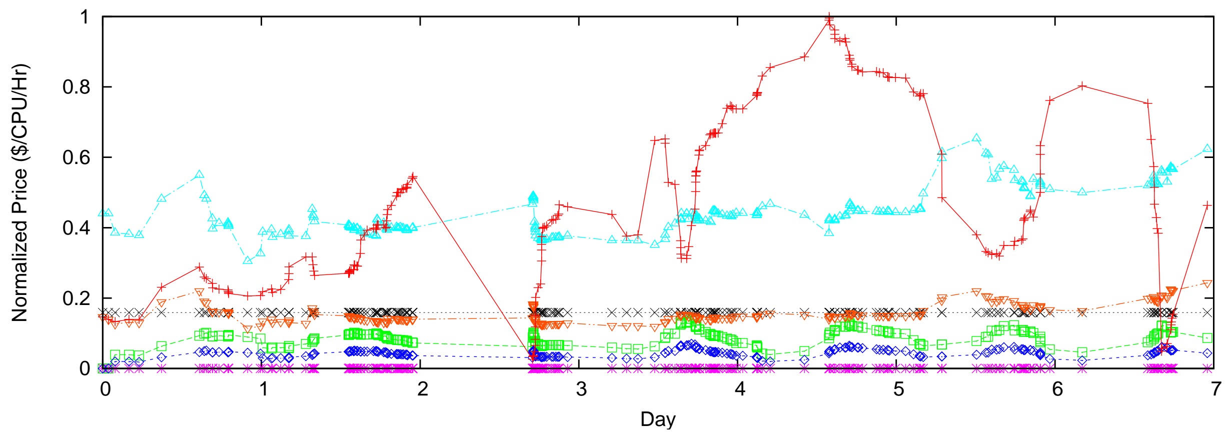


(c) Accumulated Reservations

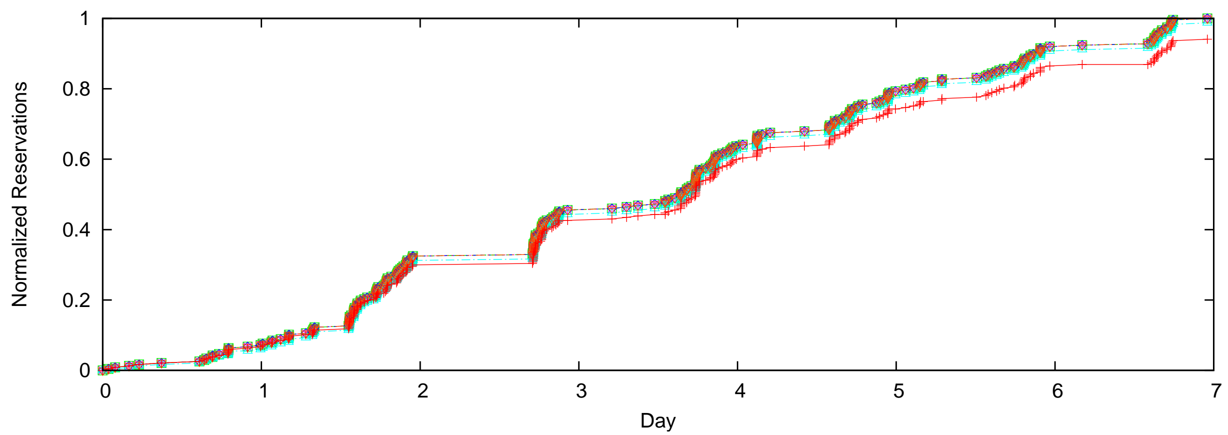
Fig. 5. Squential application requests: Low Urgency (LU).



(a) Accumulated Revenue

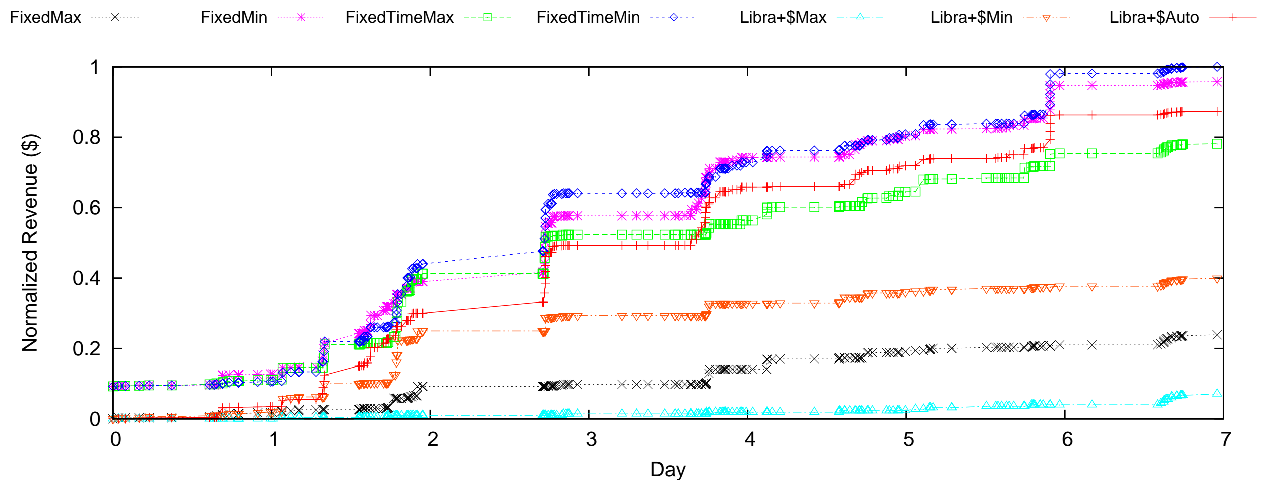


(b) Current Average Price

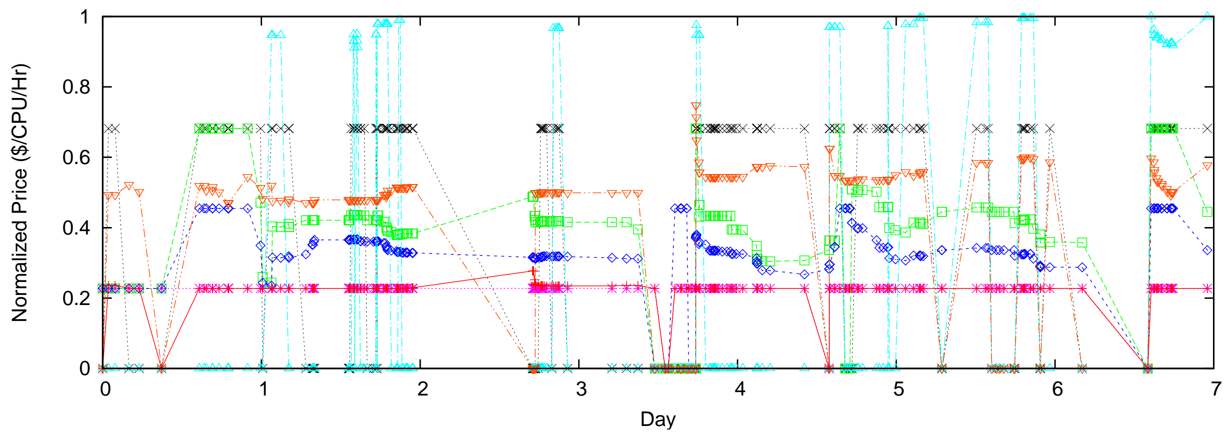


(c) Accumulated Reservations

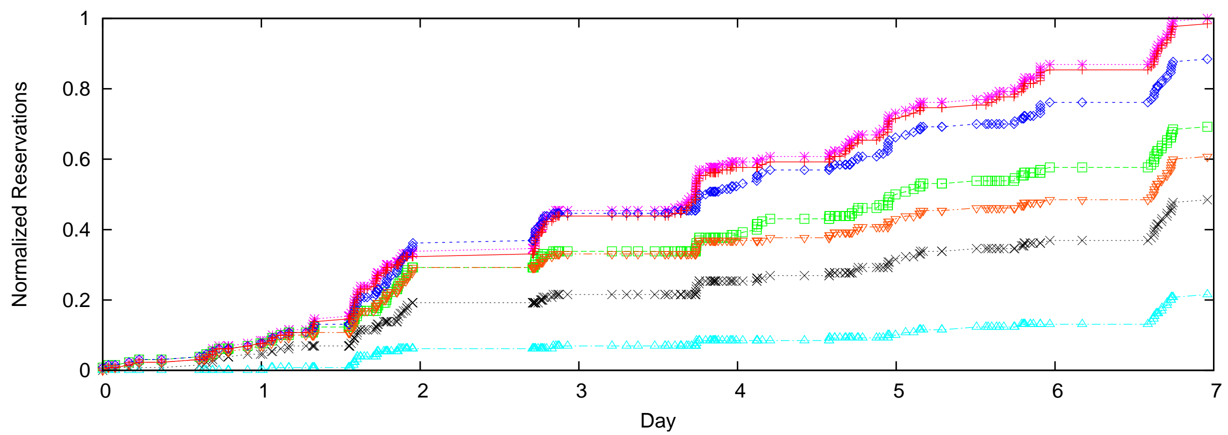
Fig. 6. Squential application requests: High Urgency (HU).



(a) Accumulated Revenue

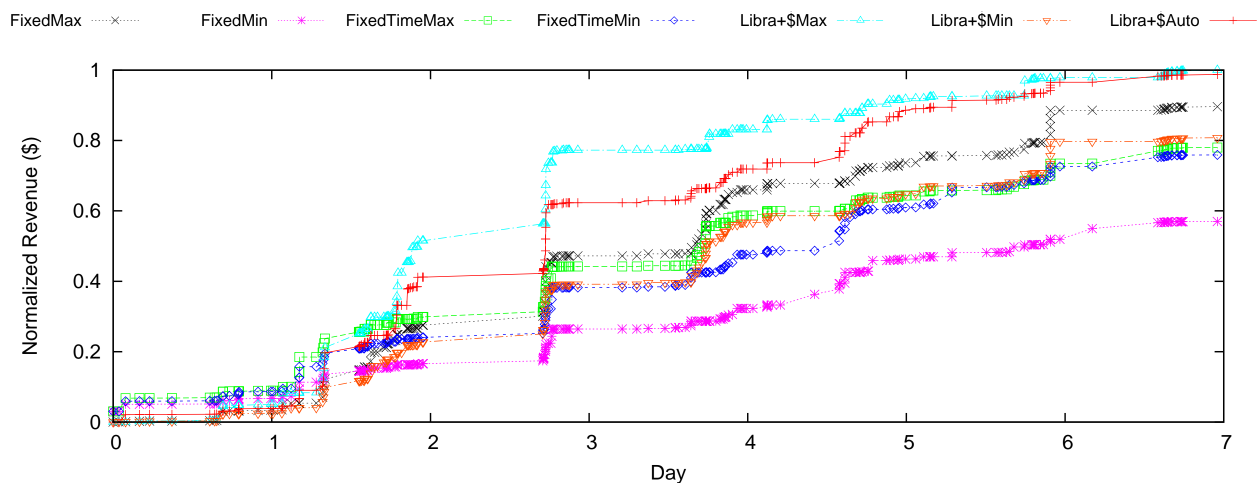


(b) Current Average Price

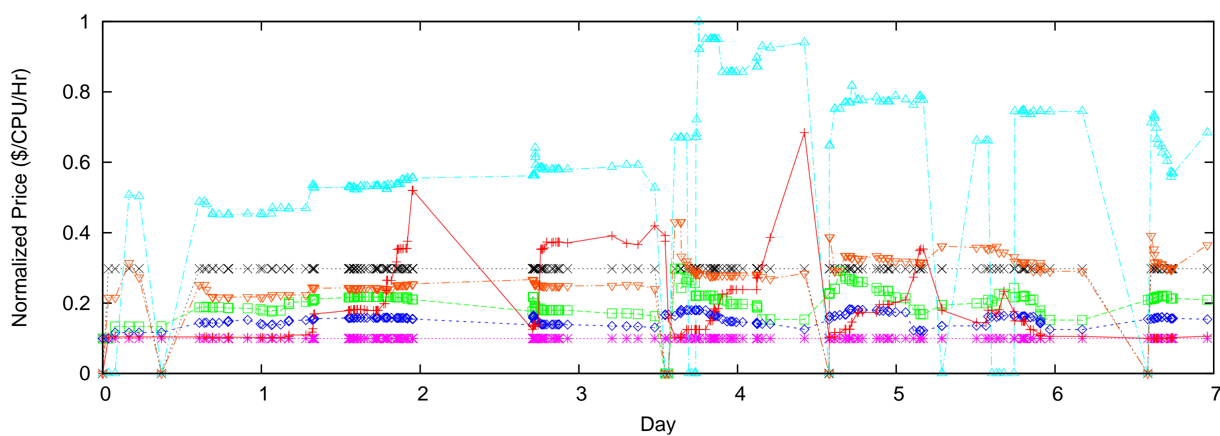


(c) Accumulated Reservations

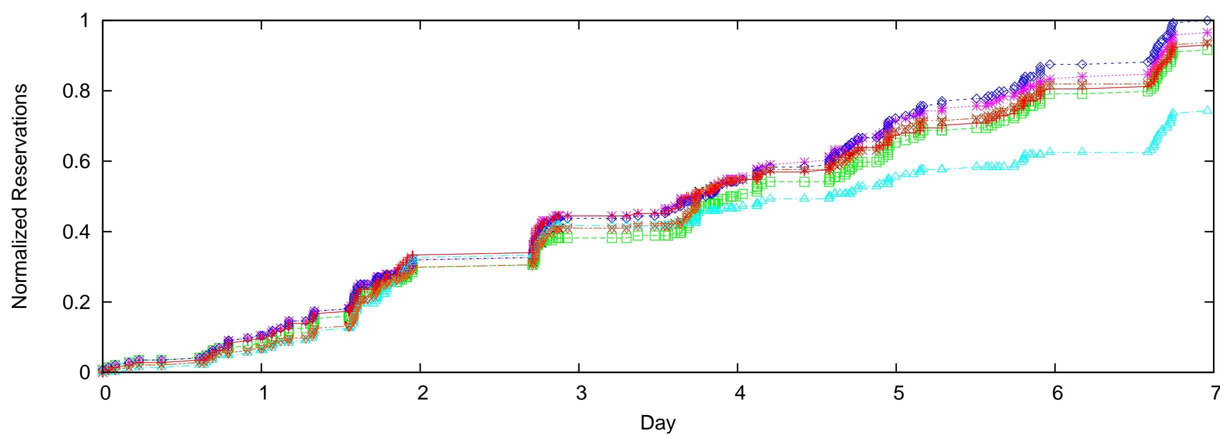
Fig. 7. Parallel application requests: Low Urgency (LU).



(a) Accumulated Revenue



(b) Current Average Price



(c) Accumulated Reservations

Fig. 8. Parallel application requests: High Urgency (HU).

The remaining FixedTimeMax and FixedTimeMin falls within the maximum and minimum bounds by charging the same price as FixedMin ($1/CPU/Hr) for off-peak (12AM–12PM), and charging either the same price as ($3/CPU/Hr for FixedTimeMax) or a lower price ($2/CPU/Hr for FixedTimeMin) than FixedMax for peak (12PM–12AM).

Given these current average price observations, one may infer that FixedMax should always provide the highest accumulated revenue (maximum bound), followed by FixedTimeMax, FixedTimeMin, and FixedMin with the lowest accumulated revenue (minimum bound). However, our performance results show that this inference is only valid for three of our tested scenarios (Figure 5(a), 6(a), and 8(a)). For LU parallel application requests (Figure 7(a)), FixedTimeMin and FixedMin provide the highest accumulated revenue, instead of FixedMax and FixedTimeMax. This is because both FixedMax and FixedTimeMax charge significantly higher current average prices (45% and 0%–45% more than FixedMin for FixedMax and FixedTimeMax respectively in Figure 7(b)) that mostly exceed the budget of LU parallel application requests, and can thus only accept a lower number of requests (52% and 31% less than FixedMin for FixedMax and FixedTimeMax respectively in Figure 7(c)). In contrast, both FixedMax and FixedTimeMax only charge current average prices that are not more than 31% higher than FixedMin for the other three scenarios (Figure 5(b), 6(b), and 8(b)). This thus demonstrates the dilemma faced by providers on how to set the best price for fixed pricing mechanisms in order to achieve the best revenue performance across all various scenarios.

We now determine whether Fixed or FixedTime is a better fixed pricing mechanism across all various scenarios. We first compare FixedMax and FixedTimeMax based on their improvement in revenue compared to FixedMin (shown in Table 2 for Figure 5(a), 6(a), 7(a), and 8(a)) because they reflect the same price difference of $2/CPU/Hr. FixedMax charges $3/CPU/Hr, while FixedMin charges $1/CPU/Hr. Likewise, FixedTimeMax charges $1/CPU/Hr for off-peak (12AM–12PM) and $3/CPU/Hr for peak (12PM–12AM), while FixedMin charges the same $1/CPU/Hr for both off-peak and peak.

Table 2 shows that FixedMax has a significantly higher standard deviation (SD = 58.01) of improvement in revenue compared to FixedMin, which is about 2.5 times more than that of FixedTimeMax (SD = 22.76). But, FixedTimeMin (which charges $2/CPU/Hr for peak) has an even lower standard deviation (SD = 8.19) than that of FixedTimeMax (SD = 22.76). This means that setting the ideal price correctly for various time periods of resource usage to satisfy different types of application and service requirements is not easy. Still, out of these four fixed pricing mechanisms, the FixedTime mechanisms are easier to derive and more reliable than the Fixed mechanisms since they support a range of prices across various time periods of resource usage and are observed to have less revenue fluctuations than Fixed mechanisms respectively.

Table 3 shows the gap in revenue from the upper bound of revenue. This upper bound is computed as the total budget of requests which are accepted. Thus, pricing mechanisms can have different upper bound values since they may not accept the same requests. Defining this upper bound enables us to know how optimal the pricing mechanisms are in terms of maximizing the revenue out of the budget given by the user. Hence, it is better to have a lower percentage gap in Table 3 since it means that the pricing mechanism is able to achieve more revenue out of the maximum budget.

Table 2
Improvement in revenue compared to FixedMin.

Pricing	**Sequential**		**Parallel**		**SD**
	LU	**HU**	**LU**	**HU**	
FixedMax	61%	26%	-72%	33%	58.01
FixedTimeMax	36%	13%	-18%	21%	22.76
FixedTimeMin	20%	7%	4%	19%	8.19
Libra+$Max	-17%	70%	-89%	43%	70.59
Libra+$Min	48%	24%	-56%	24%	45.43
Libra+$Auto	24%	87%	-8%	42%	39.65

Table 3
Gap in revenue compared to the upper bound.

Pricing	**Sequential**		**Parallel**		**SD**
	LU	**HU**	**LU**	**HU**	
FixedMax	31%	81%	23%	57%	26.36
FixedMin	75%	94%	43%	73%	21.08
FixedTimeMax	52%	87%	27%	64%	24.99
FixedTimeMin	63%	91%	33%	66%	23.75
Libra+$Max	21%	60%	14%	34%	20.27
Libra+$Min	41%	82%	26%	62%	24.46
Libra+$Auto	48%	49%	45%	52%	2.89

As listed in Table 3, all four fixed pricing mechanisms achieve a much worse percentage gap for HU requests (57%–94%) than for LU requests (23%–75%). But, FixedMin is the most optimal out of them with the lowest standard deviation (SD = 21.08), followed by FixedTimeMin (SD = 23.75), FixedTimeMax (SD = 24.99), and FixedMax (SD = 26.36). This further reinforces that FixedTime mechanisms are easier to derive and more reliable for the provider compared to Fixed mechanisms.

6.2. *Variable Prices*

We analyze the three variable pricing mechanisms which are based on Libra+$ as listed in Table 1: Libra+$Max, Libra+$Min, and Libra+$Auto. Unlike the previously discussed four fixed pricing mechanisms, Libra+$ considers the service requirement of users by charging a lower price for a request with longer deadline as an incentive to encourage users to submit requests with longer deadlines that are more likely to be accommodated than shorter deadlines. The difference in prices charged by Libra+$Max, Libra+$Min, and Libra+$Auto is primarily dependent on the β factor for the dynamic pricing component of Libra+$ as explained in Section 3.3 – a higher β factor means that Libra+$ will charge a higher price.

Table 1 shows that Libra+$Max and Libra+$Min has a β value of 3 and 1 respectively, thus Libra+$Max always charges a higher price than Libra+$Min (Figure 5(b), 6(b), 7(b), and 8(b)). However, Libra+$Max only provides higher accumulated revenue than Libra+$Min for HU requests with short deadline and high budget (Figure 6(a) and 8(a)). For LU requests with long deadline and low budget (Figure 5(a) and 7(a)), Libra+$Max instead provides the least accumulated revenue out of all seven fixed and variable pricing mechanisms, even though it charges the highest prices at various times (Figure 5(b) and 7(b)). This highlights the inflexibility of static pricing parameters to maximize revenue for different service requirements. In this case, β of Libra+$Max is set too high such that requests are rejected due to low budget.

On the other hand, Libra+$Auto is initially configured with the same pricing parameters as Libra+$Min, but will automatically adjust β based on the availability of compute nodes over time. Since Libra+$Auto does not have a statically defined β value, it has the flexibility to charge prices that are higher (Figure 6(b)) or lower (Figure 5(b), 7(b), and 8(b)) than Libra+$Max and Libra+$Min. In particular, Libra+$Auto is able to exploit the high budget of users by automatically adjusting to a higher β to increase prices and maximize revenue when the availability of nodes is low. This can be observed for HU sequential application requests (Figure 6(b)) wherein Libra+$Auto continues increasing prices to higher than that of Libra+$Max and other pricing mechanisms when demand is high such as during the later half of day 1, 2, 3, and 5.

Conversely, Libra+$Auto also adjusts to a lower β to decrease prices when demand is low to accommodate users with lower budgets. Thus, Libra+$Auto can continue to generate revenue, but at a slower rate when demand is low (i.e., when there are more unused nodes which will otherwise be wasted). This can again be observed for HU sequential application requests (Figure 6(b)), when demand is low such as during the early half of day 2, 3, 5, and 6, Libra+$Auto keeps reducing prices to lower than that of Libra+$Max to accept requests that are not willing to pay more.

With this autonomic pricing feature, we can observe that Libra+$Auto is able to generate the most highest (Figure 6(a)) and second highest (Figure 8(a)) revenue for sequential and parallel applications of HU requests respectively. In particular, Libra+$Auto is able to achieve these highest revenues by accepting an almost similar number of HU requests as most other pricing mechanisms for both sequential and parallel applications (Figure 6(c) and 8(c)). A similar number of HU requests are accepted since nodes are less likely to be available for short deadlines. Thus, it demonstrates that Libra+$Auto adjusts pricing by considering the service requirements (deadline and budget) of users.

In addition, for sequential applications, we can observe that Libra+$Auto accepts the least number of requests for both LU and HU requests (Figure 5(c) and 6(c)). But, for parallel applications, Libra+$Auto is able to accept a higher number of requests similar to most other pricing mechanisms (Figure 7(c) and 8(c)). This is because parallel applications need multiple nodes which require higher budget, compared to sequential applications which only require a single node. Hence, the low budget leads to a huge inconsistency in performance between sequential and parallel applications for other pricing mechanisms due to the inflexibility of static pricing parameters. However, Libra+$Auto is able to progressively increase the accumulated revenue

over the 7-days period for parallel applications (Figure 7(a) and 8(a)), as compared to sequential applications (Figure 5(a) and 6(a)). For example, Libra+$Auto is still able to achieve almost the same revenue as Libra+$Max even though Libra+$Max accumulates more revenue much earlier from Day 2 to 5 (Figure 8(a)). This is because Libra+$Auto adjusts to a lower β at various times to accommodate more requests with lower prices than Libra+$Max to eventually fix the initial shortfall (Figure 8(b)). Hence, unlike Libra+$Max and Libra+$Min, Libra+$Auto can also automatically adjust pricing based on application requirements, in addition to service requirements.

As listed in Table 2, Libra+$Auto has a significantly lower standard deviation (SD = 39.65) of improvement in revenue compared to FixedMin, which is about 1.8 and 1.1 times less than that of Libra+$Max (SD = 70.59) and Libra+$Min (SD = 45.43) respectively. In fact, the standard deviation of Libra+$Auto is also much lower than that of FixedMax (SD = 58.01) which is a fixed pricing mechanism. Even though Libra+$Auto has a higher standard deviation than that of FixedTimeMax (SD = 22.76) and FixedTimeMin (SD = 8.19), Libra+$Auto is able to generate considerably higher revenue for HU requests of both sequential and parallel applications by differentiating both application and service requirements of users, which is critical from the perspective of a utility computing service.

Table 3 shows that Libra+$Auto is the most optimal out of all seven pricing mechanisms across all various scenarios. Libra+$Auto has the lowest standard deviation (SD = 2.89) of gap in revenue compared to the upper bound, which is about 7.0 and 8.4 times less than that of Libra+$Max (SD = 20.27) and Libra+$Min (SD = 24.46) respectively. Unlike the other six mechanisms which have a wide range of percentage gap between LU and HU requests, Libra+$Auto consistently maintains a narrow range of percentage gap for both LU and HU requests (45%–52%). This demonstrates that Libra+$Auto is able to adjust pricing effectively to maximize revenue across all various scenarios.

However, Libra+$Auto is not the most optimal for each specific scenario. Libra+$Auto is only the most optimal for HU sequential application requests with the lowest percentage gap of 49%. Instead, Libra+$Max is the most optimal for the other three scenarios with the lowest percentage gap of 21%, 14%, and 34% for LU sequential, LU parallel and HU parallel application requests respectively. Hence, the current simple heuristic of Libra+$Auto can be further enhanced to not only maximize revenue across all various scenarios, but also for each specific scenario.

7. Conclusion

This paper studies the performance of charging fixed and variable prices for a utility computing service. Charging fixed prices is simple to understand and straightforward for users, but do not differentiate pricing to exploit different user requirements in order to maximize revenue. Hence, this paper emphasizes the importance of implementing autonomic metered pricing for a utility computing service to self-adjust prices to increase revenue. In particular, autonomic metered pricing can also be straightforward for users through the use of advanced reservations. With advanced reservations, users can not only know the prices of their required resources in the future ahead, but are also able to guarantee access to future resources to better plan and manage their operations.

Through the actual implementation of an enterprise Cloud, we show that a simple autonomic pricing mechanism called Libra+$Auto is able to achieve higher revenue than other common fixed pricing mechanisms by considering two essential user requirements: (i) application (sequential and parallel) and (ii) service (deadline and budget). The use of advanced reservations enables Libra+$Auto to self-adjust prices in a more fine-grained manner based on the expected workload demand and availability of nodes so that more precise incentives can be offered to individual users to promote demand and thus improve revenue. Experimental results show that Libra+$Auto is able to exploit budget limits to achieve higher revenue than other variable and fixed pricing mechanisms by automatically adjusting to a higher β to increase prices when the availability of nodes is low and a lower β to reduce prices when there are more unused nodes which will otherwise be wasted.

Our future work will involve conducting experimental studies using real applications and service requirements of users which can be collected by providers such as Amazon, Sun Microsystems, or Tsunamic Technologies. We also need to understand how users will react to price changes and when they will switch providers. This knowledge can be used to derive more sophisticated models to

construct a more complex autonomic pricing mechanism that considers more dynamic factors such as user response from price changes and competition from other providers. A stochastic model can then be built based on historical observation data to predict future demand and adjust prices accordingly. In addition, allowing cancellation of reservations is essential to provide more flexibility and convenience for users since user requirements can change over time. Therefore, future work needs to investigate the implication of cancellations for a utility computing service and possible overbooking of reservations to address cancellations. It may be possible to apply revenue management [8] to monitor current cancellations, amend cancellation and refund policies, and adjust prices for new reservations accordingly. The providers may also require users to pay penalties or not be entitled to any refunds for cancelling reservations depending on specific booking terms and agreements during the time of reservation.

Acknowledgments

We thank Chao Jin and Krishna Nadiminti for their help with the use of Aneka. We also want to thank anonymous reviewers for their constructive comments which helped to improve this paper. This work is partially supported by research grants from the Australian Research Council (ARC) and Australian Department of Innovation, Industry, Science and Research (DIISR).

References

[1] C. S. Yeo, R. Buyya, M. D. de Assuncao, J. Yu, A. Sulistio, S. Venugopal, M. Placek, Utility Computing on Global Grids, in: H. Bidgoli (Ed.), Handbook of Computer Networks, John Wiley and Sons, Hoboken, NJ, USA, 2007.

[2] M. A. Rappa, The Utility Business Model and the Future of Computing Services, IBM Systems Journal 43 (1) (2004) 32–42.

[3] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility, Future Generation Computer Systems 25 (2009) doi: 10.1016/j.future.2008.12.001.

[4] Amazon, Elastic Compute Cloud (EC2), http://www.amazon.com/ec2/ (Oct. 2008).

[5] Sun Microsystems, Sun Grid, http://www.sun.com/service/sungrid/ (Oct. 2008).

[6] Tsunamic Technologies, Cluster On Demand, http://www.tsunamictechnologies.com/services.htm#COD (Oct. 2008).

[7] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, A. Roy, A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation, in: Proceedings of the 7th International Workshop on Quality of Service (IWQoS 1999), IEEE Communications Society: Piscataway, NJ, USA, London, UK, 1999, pp. 27–36.

[8] R. L. Phillips, Pricing and Revenue Optimization, Stanford University Press, Stanford, CA, USA, 2005.

[9] X. Chu, K. Nadiminti, C. Jin, S. Venugopal, R. Buyya, Aneka: Next-Generation Enterprise Grid Platform for e-Science and e-Business Applications, in: Proceedings of the 3th International Conference on e-Science and Grid Computing (e-Science 2007), IEEE Computer Society: Los Alamitos, CA, USA, Bangalore, India, 2007, pp. 151–159.

[10] C. S. Yeo, R. Buyya, A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing, Software: Practice and Experience 36 (13) (2006) 1381–1419.

[11] R. Buyya, D. Abramson, J. Giddy, H. Stockinger, Economic Models for Resource Management and Scheduling in Grid Computing, Concurrency and Computation: Practice and Experience 14 (13–15) (2002) 1507–1542.

[12] G. A. Paleologo, Price-at-Risk: A Methodology for Pricing Utility Computing Services, IBM Systems Journal 43 (1) (2004) 20–31.

[13] K.-W. Huang, A. Sundararajan, Pricing Models for On-Demand Computing, Working Paper CeDER-05-26, New York University (Nov. 2005).

[14] J. P. Degabriele, D. Pym, Economic Aspects of a Utility Computing Service, Technical Report HPL-2007-101, HP Labs, Bristol (Jul. 2007).

[15] B. P. Pashigian, Price Theory and Applications, 2nd Edition, Irwin/McGraw-Hill, Boston, MA, USA, 1998.

[16] A. Sulistio, K. H. Kim, R. Buyya, Using Revenue Management to Determine Pricing of Reservations, in: Proceedings of the 3th International Conference on e-Science and Grid Computing (e-Science 2007), IEEE Computer Society: Los Alamitos, CA, USA, Bangalore, India, 2007, pp. 396–404.

[17] Y. Chen, A. Das, N. Gautama, Q. Wang, A. Sivasubramaniam, Pricing-based strategies for autonomic control of web servers for time-varying request arrivals, Engineering Applications of Artificial Intelligence 17 (7) (2004) 841–854.

[18] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, R. Buyya, Libra: A Computational Economy-based Job Scheduling System for Clusters, Software: Practice and Experience 34 (6) (2004) 573–590.

[19] C. S. Yeo, R. Buyya, Pricing for Utility-driven Resource Management and Allocation in Clusters, International Journal of High Performance Computing Applications 21 (4) (2007) 405–418.

[20] S. Venugopal, R. Buyya, L. Winton, A Grid Service Broker for Scheduling e-Science Applications on Global Data Grids, Concurrency and Computation: Practice and Experience 18 (6) (2006) 685–699.

[21] J. F. Kurose, R. Simha, A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems, IEEE Trans. Comput. 38 (5) (1989) 705–717.

[22] T. T. Nagle, J. E. Hogan, The Strategy and Tactics of Pricing: A Guide to Growing More Profitably, 4th Edition, Prentice Hall, Upper Saddle River, NJ, USA, 2006.

[23] W. Smith, I. Foster, V. Taylor, Predicting Application Run Times Using Historical Information, in: Proceedings of the 4th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 1998), Vol. 1459/1998 of Lecture Notes in Computer Science (LNCS), Springer-Verlag: Heidelberg, Germany, Orlando, FL, USA, 1998, pp. 122–142.

[24] J. Yang, I. Ahmad, A. Ghafoor, Estimation of Execution Times on Heterogeneous Supercomputer Architectures, in: Proceedings of the 22th International Conference on Parallel Processing (ICPP 1993), Vol. 1, IEEE Computer Society: Los Alamitos, CA, USA, Syracuse, NY, USA, 1993, pp. 219–226.

[25] C. Jin, R. Buyya, MapReduce Programming Model for .NET-based Distributed Computing, Technical Report GRIDS-TR-2008-15, Grid Computing and Distributed Systems Laboratory, The University of Melbourne (Oct. 2008).

[26] S. Venugopal, X. Chu, R. Buyya, A Negotiation Mechanism for Advance Resource Reservations using the Alternate Offers Protocol, in: Proceedings of the 16th International Workshop on Quality of Service (IWQoS 2008), IEEE: Piscataway, NJ, USA, Enschede, The Netherlands, 2008, pp. 40–49.

[27] Parallel Workloads Archive, http://www.cs.huji.ac.il/labs/parallel/workload/ (Oct. 2008).

[28] D. E. Irwin, L. E. Grit, J. S. Chase, Balancing Risk and Reward in a Market-based Task Service, in: Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC13), IEEE Computer Society: Los Alamitos, CA, USA, Honolulu, HI, USA, 2004, pp. 160–169.