Bidding for Highly Available Services with Low Price in Spot Instance Market

Weichao Guo[y] Kang Chen[z] Yongwei Wu[z] Weimin Zheng[z]
Tsinghua National Laboratory for Information Science and Technology (TNLIST)
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China
Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, Zhejiang
[y]gwc11@mails.tsinghua.edu.cn
[z]{chenkang, wuyw, zwm-dcs}@tsinghua.edu.cn

ABSTRACT

Amazon EC2 has built the Spot Instance Marketplace and offers a new type of virtual machine instances called as spot instances. These instances are less expensive but considered failure-prone. Despite the underlying hardware status, if the bidding price is lower than the market price, such an instance will be terminated.

Distributed systems can be built from the spot instances to reduce the cost while still tolerating instance failures. For example, embarrassingly parallel jobs can use the spot instances by re-executing failed tasks. The bidding framework for such jobs simply selects the spot price as the bid. However, highly available services like lock service or storage service cannot use the similar techniques for availability consideration. The spot instance failure model is different to that of normal instances (fixed failure probability in traditional distributed model). This makes the bidding strategy more complex to keep service availability for such systems.

We formalize this problem and propose an availability and cost aware bidding framework. Experiment results show that our bidding framework can reduce the costs of a distributed lock service and a distributed storage service by 81.23% and 85.32% respectively while still keeping availability level the same as it is by using on-demand instances.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed System—*client/server*; D.4.5 [**Operating System**]: Reliability—*fault-tolerance*; G.1.6 [**Numerical Analysis**]: Optimization—*Constrained optimization, Nonlinear programming*; G.3 [**Probability And Statistics**]: Markov processes

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.

http://dx.doi.org/10.1145/2749246.2749259.

Keywords

Distributed service; high availability; bidding framework; spot instance failure model

1. INTRODUCTION

Virtual machines are widely used in cloud computing. A running virtual machine is usually called as an instance. An instance that supports the pay-as-you-use billing schema is called as an on-demand instance on Amazon Elastic Cloud Computing (EC2). Using on-demand instances can reduce users' cost comparing to owning dedicated physical clusters in house. Many users including startup companies as well as research institutes rely on cloud computing services. On the other hand, cloud providers are willing to provide virtual machines for improving the profits while keeping constant maintaining costs of underlying physical infrastructure.

To further increase the resource utilization, Amazon EC2 proposed a new type of virtual computing instances, the spot instances. Here are the descriptions from the website of introducing spot instances: "Spot Instances allow you to name your own price for Amazon EC2 computing capacity. You simply bid on spare Amazon EC2 instances and run them whenever your bid exceeds the current Spot Price, which varies in real-time based on supply and demand." [30]. From the providers' point of view, the utilization of idle periods of the infrastructure can be further improved. For users, spot instances are much cheaper and can be used to achieve economical computing. However, as with the characteristics of spot instances, they should not be considered as always available but failure-prone despite the practical underlying hardware status. It will become hard for analysing the availability level of services built on top of spot instances.

In distributed computing, some basic services are considered as 'should be highly available'. The lock service is an example. As a fundamental building block for applications, a lock service should be always on-line. Any failure of the lock service could hurt the running of applications. Another example is the storage service, which is a fundamental building block for providing data storage. These services are usually considered critical and should be as reliable as possible. They are different from the parallel batching processing jobs which can be fixed after actual failure occurrence [24, 36, 35]. For critical services, safety property, such as a lock cannot be assigned to more than one client, must be kept at any

time. Such services do have their own fault tolerance mechanism such as Paxos-based state machine replication (SMR) [23]. The algorithms used can guarantee progress (eventually achieve the goal set forehand) if the majority number of the nodes are available for enough amount of time.

As a Paxos based mechanism can tolerate any minority of node failures in a distributed service, can we just replace normal nodes with spot instances to identically provide highly available distributed service? It is easy to just replace the on-demand instances with spot instances (maybe more spot instances) of the same distributed systems. However, it is non-trivial to analyze whether such a distributed service building on top of spot instances has the same availability level of the one bult on on-demand instances. The availability analysis becomes complicated because of the unique out-of-bid failure of spot instances. The failure probability is not fixed as it is in the traditional distributed model(usually a small constant). Thus, traditional way of using the number of available nodes for indicating the availability level should be amended by incorporating the probabilistic failure model.

To the best of our knowledge, there is no published solution to address this bidding problem, i.e., using spot instances to provide a distributed service while keeping the same availability level with another system using on-demand instances. In this paper, we will first formalize the model for describing the work of providing highly available system using spot instances. And based on this model, we propose a bidding framework to automatically make the bidding decisions for keeping appointed availability level and reducing the total cost of a distributed service.

It is feasible to achieve high availability with failure-prone spot instances when building distributed services. Take the distributed lock service as an example. The downtime of a distributed lock service using 5 geographical isolated on-demand instances should be less than 30 seconds in a whole month.[1] To achieve the same availablility level by using spot instances, we need to analyze the service availability based on the failure probability model of spot instances. As there are a large number of bidding decisions that satisfies the service availability requirement, it will be a question to decide which one is the most cost efficient. We think that this problem can be modeled as a non-linear programming problem. The objective is to minimize the total cost of spot instances for building a distributed service. The constraint is keeping the same availability level as using on-demand instances. The failure probability of a spot instance under a bid is correlative to the spot price. If the bid is lower than the spot price provided by Amazon EC2, the corresponding spot instance will not be available. As the spot prices sequence has Markovian property but the sojourn time between spot prices is not memoryless in the statistical inferences, we model the failure probability of a spot instance based on a semi-Markovian process of spot prices.

However, solving this non-linear programming is NP-hard. Exhaustive search is impractical for obtaining the optimal solution. In this paper, we have built an availability and cost aware bidding framework for obtaining a near-optimal solution practically. The framework has two main components, one is online bidding module for getting spot instances and another is spot instance failure model for estimating failure probability of spot instances. The online bidding module employs an enumeration and greedy strategy based algorithm to bid spot instances. The spot instance failure model collects the spot price data continuously, and provides the estimated failure probability of a spot instance for the next hour under a bid.

To sum up, we have made the following contributions in this paper:

- We point out that the analysis of bidding for highly available service using spot instances is different from the bidding for batching processing jobs.
- The spot instance failure model is intrinsic different from the model in traditional distributed systems. It has been formalized according to the bid and price in the marketplace.
- We have built availability and cost aware bidding framework based on the formalization of spot instance failure model and non-linear programming model. The framework is effectively applied in two different highly available distributed systems.

The remainder of this paper is organized as follows. Section 2 introduces the spot instances in Amazon EC2 and the availability of distributed services. In Section 3, we model the failure probability of spot instances and formalize the optimization problem. Section 4 describes a bidding framework to obtain a near optimal solution. Section 5 gives evaluations with the two cases to demonstrate the effectiveness of the bidding framework. Section 6 discusses the related work and Section 7 concludes this paper.

2. BACKGROUND

2.1 Amazon EC2 Spot Instance

Same as on-demand instances, spot instances give tenants a wide selection of instance types [5], which comprise varying combinations of CPU, memory, storage, and networking capacity. Users can choose the instances for their applications based on the different characteristics provided. The instances are located on an increasing number of regions all over the world [4]. Each region is in a separated geographic area and the region number keeps increasing. To achieve the greatest possible fault tolerance and stability, each region has multiple, isolated locations known as "Availability Zones", which are shown in Table 1. We suppose the highly available services are built on top of spot instances from different availability zones. Thus, the failure model of the instances is independent identically distributed. The geographical replicated configuration is widely used in highly available services, such as Spanner [16], Dynamo [17] and Azure [14].

To use a spot instance, a user should place a spot instance request that specifies the instance type, the Availability Zone, and the maximum price he is willing to pay per hour, called as a spot bid. The current default upper limit of a spot bid is four times on-demand price [6]. The

[1]According to the Service Level Agreement (SLA) from Amazon EC2, the availability of an on-demand instance will be no less than 99% or otherwise users will have 30% fee as the compensation. The 5 on-demand instances are failure independent as launching from different locations. The availability of the lock service can be calculated by subtracting the probability that 3 or more instances are simultaneously unavailable from 100%.

Table 1: Amazon EC2 Regions and Availability Zones

Region	Location	Availability Zones
US East 1	Virginia	4
US West 2	Oregon	3
US West 1	California	3
EU West 1	Ireland	3
EU Central 1	Frankfurt	2
AP Southeast 1	Singapore	2
AP Northeast 1	Tokyo	3
AP Southeast 2	Sydney	2
SA East 1	Sao Paulo	2

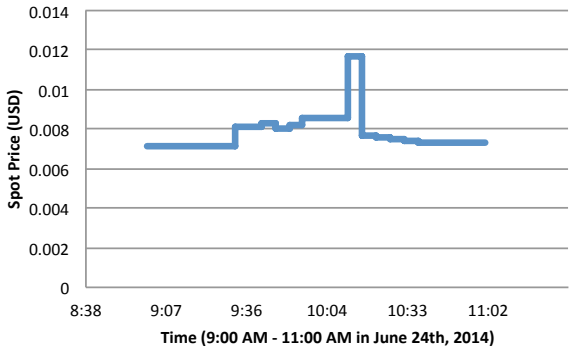

Figure 1: Spot price history for a "us-east-1a.linux.m1.small" spot instance

spot price is set by Amazon EC2, and fluctuates according to the supply and demand of the spot instance capacity. Figure 1 shows some spot price fluctuations for "us-east-1.linux.m1.small" instances during 9:00 AM - 11:00 AM on June 24th 2014. When a bid exceeds the spot price, the spot instance is launched and will run until the spot price rises above the bid(out-of-bid failure) or the user chooses to terminate it. If the spot instance is terminated by Amazon EC2, the user will not be charged for any partial hour of usage. However, if the user terminates the spot instance, he will pay for any partial hour of usage as he would in occasions like using on-demand instances. Spot instances are charged with the spot price. Obviously, a higher spot bid returns a more reliable and available spot instance, and may also induce a higher charge.

2.2 The Availability of Distributed Service

To achieve high availability, state machine replication (SMR) is a general way for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. Each operation should be a consensus value decided by SMR. Paxos [23] has been proved to be an effective consensus protocol in SMR to build highly available distributed services. Paxos family protocols have been widely used in varies of distributed services [13, 10, 26].

In fact, Paxos is one of the quorum based techniques [18] for building a highly available distributed system. Such protocols often use 'vote' like algorithms. A request has to obtain sufficient votes v by the nodes in the distributed system to perform an operation. In this context, the distributed system will be unavailable when the live nodes have insufficient votes. A minimal number of v is called a quorum. If a quorum of nodes is available, the distributed system is available otherwise the system will be considered as unavailable.

Service availability can be defined as the probability that a request can get an appropriate response according to the specification. The availability is determined by the failure probabilities of nodes in the system. The availability of quorum systems is studied in [27]. For convenience, we introduce a definition of *Acceptance Set* [2] .

DEFINITION 1. *A collection $\mathcal{A}$ of sets over a nite universe U representing the nodes of a distributed system is called an acceptance set if*

1) $S \cap T \neq \emptyset$ for all $S, T \in \mathcal{A}$. (Intersection)

2) If $S \in \mathcal{A}$ then $T \in \mathcal{A}$ for all $T \supseteq S$. (Monotonicity)

The collection of minimal quorums is $\mathcal{S} = \mathcal{S}(\mathcal{A}) = \{S \in \mathcal{A} \mid S \setminus \{u\} \notin \mathcal{A}$ for all $u \in S\}$.

Let $\mathcal{A}$ be an acceptance set of a distributed system, for each set $S \in \mathcal{A}$, the probability of which the elements of S are alive and the rest are in failure is

$$\prod_{i2S}(1-p_i)\prod_{j2\overline{S}}p_j,$$

where $\overline{S}$ denotes $U \setminus S$, $\mathbf{p} = (p_1, p_2, \cdots, p_n)$ denotes the failure probabilities of nodes in the distributed system over a period of time.

As S are different sets, the non-failure probability (the "availability") of $\mathcal{A}$ can be further extended as

$$A_A = \sum_{S2A}(\prod_{i2S}(1-p_i)\prod_{j2\overline{S}}p_j) \quad (1)$$

Here, we introduce another definition of *Optimal Availability Acceptance Set*.

DEFINITION 2. *An acceptance set $\mathcal{A}$ over a nite universe U representing the nodes of a distributed service is called an optimal availability acceptance set if*

1) $A_A \geq A_B$ for all acceptance sets $\mathcal{B}$ over the same universe U.

For a distributed system, the non-failure probability of the optimal availability acceptance set $\mathcal{A}_o$ is equivalent to the expected availability of the distributed service. The acceptance set discussed here assumes the indepedent failure of spot instances. This is consistent to our discussion before that the highly available services are built over instances from different availability zones. As we have to estimate the availability of distributed service, acceptance set will be a constraint in the non-linear programming studied below.

3. PROBLEM FORMALIZE

Since configuring Paxos based SMR in a distributed system can tolerate a minority of node failures, it seems that we can replace the normal nodes in a distributed system with spot instances directly. In fact, the original high availability of the distributed service may no longer exist any more. This can be illustrated in the following example.

Supposing that a Paxos based distributed system has 5 nodes. The failure probability of each node is 0.01. This

distributed system can tolerate any two-node simultaneous failures. According to Equation (1), the expected availability of the distributed service is 0.9999901494, which means that there should be only about 25.5 seconds downtime in one month. If replacing all the 5 nodes with Amazon EC2 spot instances and setting the bids same to spot price, a same availability level can not be achieved. Although the replaced distributed system can still tolerant the same number of node failures as before, the non-failure probability of the distributed system is much less than the original one. Taking 5 spot instances from different Amazon EC2 availability zones in June, 2014 as an example, the spot prices of availability zone US East 1a, US East 1c, US West 1b, US West 2a, US West 2b are $ 0.008, $ 0.008, $ 0.009, $ 0.009, $ 0.009 at 0:00 AM on June 1st, 2014 respectively. If we bid a spot instance in each the availability zone with the aforementioned spot prices, node failures are encountered more often in such a distributed system. The downtime of the distributed service in June, 2014 may be more than 1500 seconds.

In essence, the number of tolerating simultaneous node failures with spot instances does not indicate the same availability with on-demand ones for a distributed system. Because the failure probability of spot instances are usually much higher than that of normal nodes. The failure values are in fact constantly changing with the fluctuation of spot prices. Therefore, we model the failure of spot instances from the spot bid and spot price.

Based on the failure model, we have to address the bidding decision problem i.e. how to keep a distributed service staying highly available and how to bid to minimize the cost of spot instance. We apply a non-linear programming to solve this optimization problem.

3.1 Spot Instance Failure Model

The availability of a distributed system is based on the availability of each component in the system. Considering Amazon EC2 instances, the availability of an instance can be estimated by its failure probability. Furthermore, let $MTBF$ (Mean Time Between Failures) denotes the average time between failures of an instance, $MTTR$ (Mean Time To Repair) denotes the average time for an instance to recover from a failure, the availability A of an Amazon EC2 instance can be measured as following.

$$A = \frac{MTBF}{MTBF + MTTR} \tag{2}$$

Diverse causes can bring down an Amazon EC2 instance including software and hardware errors. According to Amazon EC2's SLA [7], the measured availability of an on-demand instance is about 0.99. It means that the failure probability of an on-demand instance is about 0.01.

For a spot instance in Amazon EC2, the new and major type of failure is the out-of-bid failure as discussed in section 2.1. An out-of-bid failure is caused by the spot price fluctuates above the bid of the spot instance. Considering this type of failure only, the failure probability of a spot instance at time t can be represented by

$$Pr(p(t) > b) \tag{3}$$

Where $p(t)$ denotes the price at time t, and b denotes the bid.

As other kinds of failures are independent with the out-of-bid failure of Amazon EC2 spot instances, and the availability of a spot instance without out-of-bid failures will be the same as an on-demand instance, the failure probability of a spot instance $FP(t)$ can be further represented by

$$FP(t) = 1 - (1 - FP^0) \cdot (1 - Pr(p(t) > b)) \tag{4}$$

Where FP^0 denotes the failure probability of a corresponding on-demand instance. Here $FP^0 = 0.01$.

The total failure time of a spot instance is exactly the cumulative time of out-of-bid failures. Thus the failure probability of a spot instance in a time duration $d, d > 0$ can be further represented as

$$\int_0^d FP(t)dt \tag{5}$$

Amazon EC2 assigns spot instances to bidders in descending order of their bids until all available spot instances have been allocated or all spot instance requests have been satisfied. The spot price is equal to the lowest winning bid. And after a period of time, the spot price may or may not change depending on the changing of demand and supply. Figure 1 depicts the spot price history during 9:00AM - 11:00AM on June 24th, 2014 for the "us-east-1.linux.m1.small" spot instances. The spot price first remains at $0.0071 before switching to $0.0081, and reaches up to $0.0117 after about half an hour. Thus, we should estimate the failure probability of a spot instance based on the spot price variations.

As shown in Figure 1, the spot price remains at a fixed value S_i for a time duration d_i before switching to another value S_{i+1}. In essence, spot price sequence $(S_i, i = 1, 2, \cdots, n)$ and sojourn time sequence $(d_i, i = 1, 2, \cdots, n)$ are both random process. Previous works [15] have proved that the spot price sequence has Markovian property with investigating the Chapman-Kolmogorov equation [19]. And the sojourn time sequence can be modeled as a temporal point process [12].

As the sojourn time between spot prices is not memoryless, we characterize the spot price variations by a semi-Markovian chain model, which is also employed in [31]. Denote the set of all unique spot prices as $\mathcal{S}$, where $\mathcal{S} = \{s_i, i = 1, \cdots, |\mathcal{S}|\}$, and denote the state space of sojourn time as $\mathcal{T}$, where $\mathcal{T} = \{\tau_i, i = 1, \cdots, |\mathcal{T}|\}$. The stochastic kernel of the semi-Markovian chain can be represented as

$$Q(i, j, k) = (q_{i,j,k}; s_i, s_j \in \mathcal{S}, k \in \mathcal{T}) \tag{6}$$

where

$$q_{i,j,k} = Pr(S_{n+1} = s_j, S_n = s_i, \tau_n = k) \tag{7}$$

i.e., the probability that at current state i, a transition will happen to state j after time k.

The detail statistical inference is in [31]. With this spot price model, we can calculated the transition probability of two spot prices in future, and then estimate the failure probability of a spot instance under a specific bid.

3.2 Cost Minimization Problem

We apply a non-linear programming model to this bidding problem. The objective is to minimize the cost of spot instances when building a distributed service. The constraint is that the availability of the distributed service built with spot instances is not worse than the one built with on-demand instances. We estimate the availability of

the distributed service based on the spot instances failure probability estimation.

According to the spot pricing rules of Amazon EC2 mentioned in section 2.1, users need no payment for the partial hour of a spot instance that is terminated by Amazon EC2. If a user can precisely predict the price changing of a spot instance, it is possible to exploit this feature to reduce the cost or even free computation. However, it is difficult to take advantage of out-of-bid failure as an accurate prediction is required to notify when the price will change and what price it will be. Here we are not going to harness such Amazon EC2's spot pricing rule so as to simplify the cost minimization problem.

In Amazon EC2, not only varies of hardware or software failures of instances are isolated by different availability zones, but also the out-of-bid failure is isolated by pricing isolation of different availability zones. To ensure the failure independence of spot instances we bid, the distributed service should use only 0 or 1 instance in each availability zone as mentioned before. There are more than 20 availability zones in Amazon EC2. This is large enough for choosing moderate number of participants in a Paxos group in practical systems(usually 5 or 7).[13]. And performance requirements can be satisfied by launching multiple Paxos groups.

As Amazon EC2 charges for a spot instance hourly, the bidding interval should be n (a positive integer) hours. In each bidding interval, the cost minimization problem can be formalized using a non-linear programming. The decision variables in this model are the spot instance bids for different availability zones. Bidding decisions can be denoted as a variable vector $\mathbf{b} = (b_1, b_2, \cdots, b_n)$. The bid for a spot instance in the i-th Availability Zone is b_i, $i = 1, 2, \cdots, n$.

Unfortunately the cost for a spot instance in the next interval is still unknown when bidding. This is due to the way Amazon EC2 charges. Amazon EC2 charges hourly for a spot instance with the last price of a spot instance in the hour rather than the bid. For a system consist of multiple spot instances, the goal is to achieve minim the sum cost of each instance. As the spot prices in a future time point are random variables, we can minimize the expectation cost of spot instances under the given bids instead of minimizing the unknown cost. But the actual cost may be much higher than the expectation in some bidding intervals. We choose the sum of bids, an upper bound of the cost, instead of the expectation cost as the objective function in the non-linear programming.

The key constraint in the non-linear programming is to ensure that the availability of a distributed service with spot instances is comparable to that with on-demand ones. The availability of a distributed service can be represented as the availability of its optimal availability acceptance set. Denote the optimal availability acceptance set of a distributed service S as $\mathcal{A}_o(S, \mathbf{FP})$, $\mathbf{FP}$ is the node failure probability vector. Denote a distributed service built with spot instances as S_s, the spot price of availability zone i as p_i, and the failure probability vector of the spot instances under bids $\mathbf{b} = (b_1, b_2, \cdots, b_n)$ in S_s as $FP(\mathbf{b})$. Similarly, denote the associate distributed service built with on-demand Instance as S_o, the number of on-demand instances in S_o as m. We finally can formalize the cost minimization problem as

$$\min \sum_{i=1}^{n} b_i \tag{8}$$

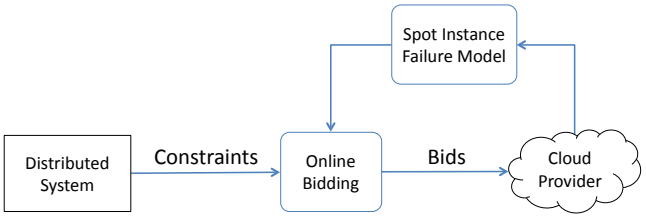

Figure 2: The bidding framework

s.t.

$$\sum_{i=1}^{n} \epsilon(b_i - p_i) \geq m \tag{9}$$

and

$$A_{A_o(S_o, \mathbf{FP}')} - A_{A_o(S_s, FP(\mathbf{b}))} < \varepsilon \tag{10}$$

The inequality (9) is a basic constraint that keeps the nodes are online at first to ensure that the distributed service built with spot instances initializes correctly. In the inequality (10), ϵ(u) equals 1 if $u > 0$ and 0 otherwise, and ε represents as an infinitesimal. It can be set to an acceptable availability variation in practice, e.g. 0.000001.

4. BIDDING FRAMEWORK

Solving the non-linear programming is NP-hard. All the possible candidates of bids $\mathbf{b}$ need to be traversed and verified with the availability constraint. The size of traverse space is m^n, where m is the number of possible prices and n is the number of availability zones. Using exhaustive search is not practical.

Fortunately, we just aim to reduce the cost of spot instances and do not have to get the optimal solution. As solving this optimization problem in an acceptable short time is impossible, we seek for a near-optimal solution rather than the optimal one. We have built a cost and availability aware bidding framework to make practical solutions. As illustrated in Figure 2, bidding decisions are made by online bidding module at the beginning of each bidding interval. And then the spot bids are issued to the cloud provider i.e. Amazon EC2. The spot instance failure estimation is used to estimate the failure probability of a spot instance for the next bidding interval under a specific bid. The estimated failure probability of a spot instance is then used to verify the availability requirement of the distributed system. With more and more spot prices data collected, the spot instance failure probability estimation can be improved.

In the bidding framework, the bidding decisions will change between two bidding intervals if spot prices fluctuate drastically. Some spot instances should be replaced by some new ones in different availability zone. To keep safety, i.e. the service availability level, the new spot instances are launched before the next bidding interval starts and added to the system, then the old spot instances that should be replaced are removed from the system at the beginning of the coming bidding interval. Adding and removing a spot instance is supported by the *view change* of Paxos. The startup time of a spot instance is usually 200∼700 seconds and mainly varies in regions [25]. The actual time of a bidding interval is shortened by the startup time of spot instances. We give a discussion about the different choices of bidding interval in section 5.5.

4.1 Online Bidding

As we have illustrated, the failure probabilities of nodes are fixed to a same small constant in a traditional distributed system model. A quorum is a simple majority of the nodes in the distributed system. For a distributed system with nodes that have different failure probabilities, a calculated weighted voting assignment has been proved to be the optimal configuration. The optimal availability vote assignments for weighted voting mechanism are well studied in [33, 32, 2]. Denote all the failure probabilities of n nodes as $p_i, i = 1, 2, \cdots, n$, it has been proved that the optimal quorum system is a monarchy with one of the least unreliable processors as the king if $p_i \geq \frac{1}{2}$ for all i, and any node i with $p_i > \frac{1}{2}$ should be a dummy if only parts of nodes with $p_i \geq \frac{1}{2}$ in [2]. If $0 < p_i < \frac{1}{2}$ all i, the optimal weights are defined by the formula in [32, 33]

$$w_i = log_2(\frac{1 - p_i}{p_i}) \tag{11}$$

However, in a practical scenario, Formula (11) cannot be used in the case where the differences of p_i are significant. For example, suppose that the failure probabilities of nodes are respectively 0.01, 0.1, 0.1 in a distributed system with three nodes. Depending on Formula (11), the node with failure probability of 0.01 has a dominated vote that is larger than the sum of the other two nodes' votes. Ideally, this is reasonable because we should use a monachy system when one node is much more reliable than the other nodes. But in practical, the failure probability of a node is measured in a finite time period, and has a deviation away from the ideal value representing the steady state when the time period is infinite. The out-of-bid failure in the spot instance failure model just fits this case. Furthermore, some Paxos family protocols are designed without considering weighted voting assignment mechanism. To keep the bidding algorithm simple and compatible, we still use a simple majority of nodes as a quorum.

As the spot instances are fixed to an equal weighted vote, only same failure probabilities of spot instances meet the definition of optimal configuration. In our online bidding algorithm, we try to make failure probabilities be closed to each other. And enumeration and greedy strategy are used in the algorithm.

The pseudo code of the online bidding algorithm is shown in Figure 3. The algorithm gets the configurations of a distributed system as input, including the system availability requirement and instance type. The procedures of online bidding are simple: 1) For all possible number of nodes n, calculates the failure probability FP that satisfies the system availability requirement when each node has the same failure probability. 2) Under the configuration of n nodes, for all availability zones, gets the minimal bid of which the estimated failure probability is less than FP. The estimated failure probability depends on the spot instance failure model of the availability zone, the bid, the spot price and the sojourn time of the spot price. 3) Comparing the value of the bids, selects the availability zones in a greedy way. 4) By accumulating the selected bids, calculates the upper bound of the cost under each configuration of n nodes and returns the bids that have the lowest upper bound of the cost as the bidding decision. This algorithm does not always lead to an optimal bidding decision, but obtain a good and near optimal solution in practise.

Algorithm:Bidding

Input: *availability*, *type*
Output: *bids*

```
1: zones ← get_availability_zones()
2: n ← 1
3: while n ≯ zones.length do
4:     FP ← node_failure_pr(n, availability)
5:     for all zone ∈ zones do
6:         spotprice ← get_spot_price(zone, type)
7:         bid ← spotprice.price
8:         while bid ≯ get_on_demand_price(zone, type) do
9:             if estimate_FP(zone, bid, spotprice) ≯ FP then
10:                break
11:            else
12:                bid ← bid + 1
13:        bids[n][zone] ← bid
14:    sort(bids[n])
15:    cost_upper_bound[n] ← sum(bids[n][1 : n])
16:    n ← n + 1
17: m ← min_key(cost_upper_bound)
18: return bids[m][1:m]
```

Figure 3: Bidding algorithm

4.2 Failure Probability Estimation

In the bidding framework, the spot instance failure model is employed by the online bidding module. To estimate the failure probability of a spot instance, a spot price model is embedded in the spot instance failure model. We use the semi-Markovian chain to model the spot price sequence as mentioned in section 3.1. Thus, a key task of spot instance failure probability estimation is to reconstruct the transition distribution from the observed spot price history data.

Wee [34] shows that the spot price sample cumulative distribution functions have significant increase around every hour in 2011. However, the spot prices data we collected in 2014 shows that the spot price change frequency has raised to many times each hour. For simplicity, we set the time unit of the semi-Markovian chain to 1 minute. The sojourn time d_i in the sample data is discretized as

$$\tau_i \triangleq \tau(S_i \to S_{i+1}) \triangleq \lfloor d_i \rfloor \tag{12}$$

We use an empirical estimator, which resembles a Maximum Likelihood Estimator (MLE) essentially [9]. The stochastic kernel $\mathbf{Q}$ is reconstructed by:

$$\widehat{q_{i,j,k}} = \frac{N_{i,j}^k}{N_i}, if N_i \neq 0 \tag{13}$$

otherwise, $\widehat{q_{i,j,k}} = 0$.

Where N_i denotes the number of occurrences of price $s_i \in \mathcal{S}$, $N_{i,j}^k$ denotes the number of transitions from price $s_i \in \mathcal{S}$ to $s_j \in \mathcal{S}$ with sojourn time of $k \in \mathcal{T}$.

Although we can make a bid for a spot instance as high as four times the on-demand price, we should choose an on-demand instance rather than a higher bid for a spot instance to avoid the potential higher charge. Thus, we force the bid of a spot instance lower than the corresponding on-demand

instance in our bidding framework. In a time unit, the failure probability of a spot instance under a bid b as

$$FP(b) = \begin{cases} 1 & \text{if } 0 \le b \le p \\ 1 - (1 - FP^0) \cdot \sum_{j=p}^{b} \widehat{q_{p,j,k}} & \text{if p < b < o} \end{cases} \quad (14)$$

where b denotes a bid, p denotes the spot price, k denotes the sojourn time, o denotes the corresponding on-demand price, FP^0 denotes the failure probability of the corresponding on-demand instance, which is fixed to 0.01.

The failure probability of a spot instance in a bidding interval is the failure probability expectation of each time unit, which is a discretization of Formula (5).

5. EVALUATION

We have implemented a prototype of the bidding framework in Python. The prototype interacts with Amazon EC2 via boto [11], which is a Python interface library to Amazon Web Services. To evaluate our bidding framework, we apply it to a distributed lock service and a distributed storage service on Amazon EC2. The experiments include a micro-benchmark, a one-week-long running on Amazon EC2 for feasibility verification and two 11-week-long trace replays for cost and availability evaluation.

5.1 Experimental Systems

5.1.1 Distributed Lock Service

A distributed lock service is intended for large-scale loosely coupled distributed systems. A representative distributed lock service is Google Chubby [13], which can help thousands of nodes to synchronize their activities and to agree on basic information about their environment such as system members.

Chubby provides an interface much like a file system with advisory locks and uses the Paxos protocol for practical distributed consensus to achieve high availability. A Chubby server is usually configured with 5 replicas. The Chubby clients communicate with Chubby server using a client library via RPC. The quorum in such a 5-node distributed system is a simple majority. Chubby follows the assumption here that the replicas are replicated in different regions i.e. similar to availability zones in Amazon EC2.

5.1.2 Erasure Code Based Distributed Storage Service

Distributed storage services usually provide an object store or key value store for clients while tolerating a portion of machine restarts and even permanent machine or disk failures. Using a distributed storage service across availability zones can provide better availability for tolerating data center crash or un-reachable problem. Instead of primary-backup techniques, Gaios [10] and Megastore [8] have employed Paxos based SMR for fault tolerance in the implementation of distributed storage service.

Erasure code [28] is a forward error correction (FEC) code in information theory and originally used to recover messages from independent packets loss in network transmission. And it has been widely adopted in distributed storage systems [21, 29] for reducing storage and network cost. In a common form of erasure coding, the original data object will be firstly divided into m equal-sized chunks and then k parity chunks of the same size will be generated. The total number of chunks is $n = m + k$. The erasure code algorithm can reconstruct the original data from any m chunks out of total n chunks. This erasure coding can be denoted as $\theta(m, n)$.

Recently, Mu et al. [26] has proposed a Paxos based protocol, RS-Paxos, which can do erasure coding correctly in distributed services without the assumption of a "synchronous" network model. RS-Paxos can be employed in a distributed storage service for network transmission and disk writes reduction by sending coded data shards instead of full copy. We call such a distributed service as an erasure code based distributed storage service.

A standard configuration for RS-Paxos is also 5 nodes and a $\theta(3, 5)$ erasure coding. Notice that RS-Paxos can only tolerate one-node failure instead of two-node failures, which is very different from a distributed lock service. This is because the write quorum of the service is different from the one using a replication mechanism. To guarantee reconstruction of the original data, the intersection size of the acceptance set should be 3 instead of 1.

5.2 Experimental Setup

These two distributed services built with on-demand instances are set as the baseline in our experiments. Although using reserved instances can reduce $30\% \sim 40\%$ cost at most, it is inflexible and difficult to adapt to the changing of service load. The failure probability of an on-demand instance is 0.01 as illustrated in [7]. The distributed lock service used in the experiments is a simple implementation based on Paxos, and the distributed storage service is an implementation of RS-Paxos [26]. We configure the two systems both with 5 on-demand instances, which is the common configuration in practical systems [13, 26]. In such a configuration, the distributed lock service can tolerate any two-node failures. The distributed storage service can tolerant only any one-node failures instead of two-node failure as illustrated in section 5.1.2.

For comparison, we also pick a heuristic bidding strategy. In this strategy, the availability zones with the lowest $n + m$ spot prices are chosen, assuming that there are n nodes in the original distributed system configuration, m is the number of additional nodes. In these experiments, n is 5. For each availability zone, a spot instance bid is set as the spot price with an extra portion p, such as 10% or 20%. As there are various of selections of m and p, we use some typical configuration of m and p in the comparison experiments. For simplicity, we denote such a strategy with m additional nodes and p extra portion of bid as $Extra(m, p)$, and denote our bidding algorithm and framework as "Jupiter".

The experiments are run over 17 availability zones of Amazon EC2. The distributed lock service is built with the "linux.m1.small" instances. Each "linux.m1.small" on-demand instance is charged about \$ $0.044 \sim 0.061$ for one hour. The erasure code based distributed storage system is built on the "m3.large" instances, which have more CPU and memory capacity than "m1.small" ones. Each "linux.m3.large" on-demand instance is charged about \$ $0.14 \sim 0.201$ for one hour.

For each availability zone, the spot instance failure model is first trained with about three months of spot prices data. Such data training leads to convergence. In the experiments, we run the bidding framework for one week on Amazon EC2

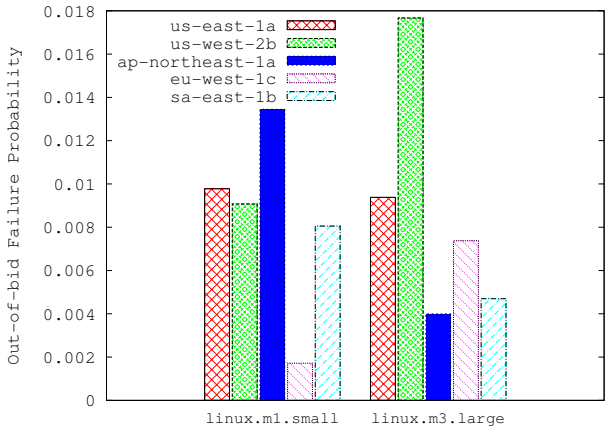

Figure 4: Measured out-of-bid failure probability of a spot instance under the estimation failure probability of 0.01

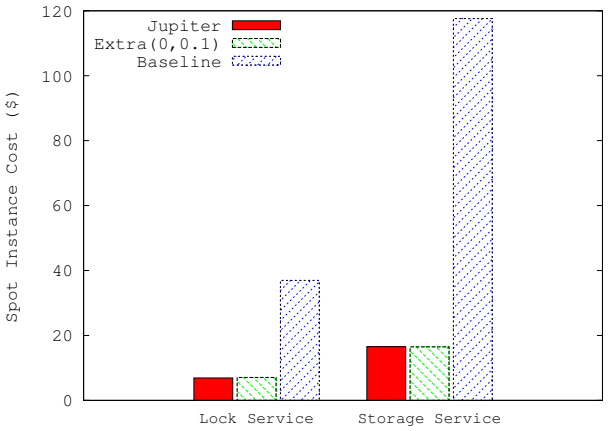

Figure 5: Spot instance cost of a distribued service under different bidding strategies from December 15th, 2014 to December 22th, 2014

and replay two 11-week-long spot price trace according to the Amazon EC2's spot pricing rule. The startup time of a spot instance is considered in the replay. It mainly varies in regions and usually 200∼700 seconds according to [25].

5.3 Micro-Benchmark

For the spot instance failure model, our major concern is the precision of spot instance failure probability estimation. Accordingly, we compare the measured failure probability of a spot instance and its expected failure probability in the spot instance failure model.

In this test, we first make a bid for each availability zone to keep the probability of out-of-bid failure no more than 0.01 in one month based on the spot instance failure model. Then we examined the out-of-bid failure probability by comparing the bids with the month's spot prices data. Figure 4 shows the measured failure probability of 5 availability zones.

For both "linux.m1.small" and "linux.m3.large" spot instances, the measured out-of-bid failure probability is less than 0.01 in most cases. There are two exceptions in all the test cases. One is in availability zone 'ap-southeast-1a', the out-of-bid failure probability is about 0.013553 for a 'linux.m1.small' instance. The other one is in availability zone 'us-west-2b', the out-of-bid failure probability of 'linux.m3.large' is about 0.017665. The test results show the spot instance failure model can estimate failure probability with minor deviation. And the failure probability estimation of spot instances can be more accurate with new spot prices data.

5.4 Feasibility

In December 2014, we performed a one-week-long running of the bidding framework on Amazon EC2. Our bidding framework functioned correctly and kept the two distributed services always available in the one-week experiment. The heuristic strategy $Extra(0, 0.1)$ is also tested in this experiment. The interval of each bidding is set to one hour in both strategies.

As shown in Figure 5, the cost of the distributed lock service under our bidding framework is about $ 6.91, only one sixth of the baseline and a little lower than the cost under $Extra(0, 0.1)$. Both are always available in the one-week-long running. The cost of the distributed storage service under our bidding framework is about $ 16.53, close to the cost under $Extra(0, 0.1)$. The distributed storage service encountered no failure for one week. However, the distributed storage service under $Extra(0, 0.1)$ failed in the running. Our bidding framework outperformed the $Extra(0, 0.1)$ strategy on the whole. This experiment showed that our bidding framework is practical for reducing cost while still keeping the service highly available.

5.5 Cost and Availability

We evaluated our bidding framework in a long term by replaying the spot prices trace. As cost and availability of a spot instance are certained with the given spot prices data, the result is the same as real running the bidding framework on Amazon EC2. Two strategies, $Extra(0, 0.2)$ and $Extra(2, 0.2)$, are introduced for comparison. There are 11 weeks of "linux.m1.small" spot prices data and 11 weeks of "linux.m3.large" spot prices data from October 2014 to December 2014 in the trace replays. The distributed lock service is replayed with "linux.m1.small" trace and the distributed storage service is replayed with "linux.m3.large" trace. We mainly focus on the cost and availability of the two building cases. The bidding intervals of 3, 6, 9, and 12 hours are also tested besides 1 hour.

The cost of a distributed lock service for 11 weeks with 5 "linux.m1.small" on-demand instances in the cheapest availability zones is $406.56. For the erasure code based distributed storage service with 5 "linux.m3.large" on-demand instances, the value is $1293.6. These are the baselines in this experiment. The cost of distributed services under heuristic bidding strategies and our bidding framework are shown in Figure 6 and 8. The availability results are shown in Figure 7 and 9.

As shown in Figure 6, our bidding framework only cost about one-fifth of the baseline in the distributed lock service for the best case. The cost of our bidding framework with the bidding interval of 6 hours outperformed all the other solutions. In this case, the cost is about $ 77.3. For the bidding intervals of 1, 9, 12 hours, the cost of our bidding

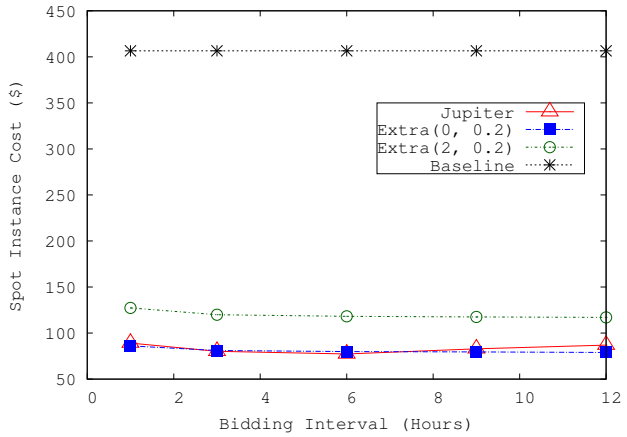

Figure 6: Spot instance cost of a distributed lock service under different bidding strategies from October 2014 to December 2014

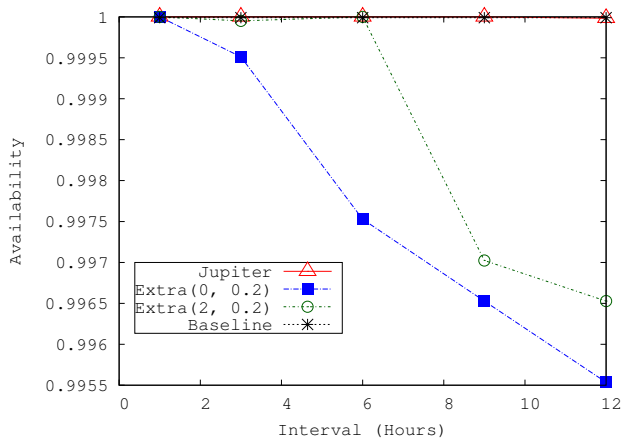

Figure 7: Availability of a distributed lock service under different bidding strategies from October 2014 to December 2014

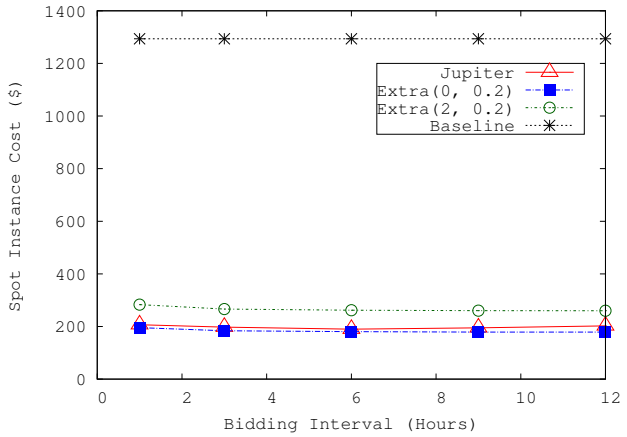

Figure 8: Spot instance cost of an erasure code based distributed storage service under different bidding strategies from October 2014 to December 2014

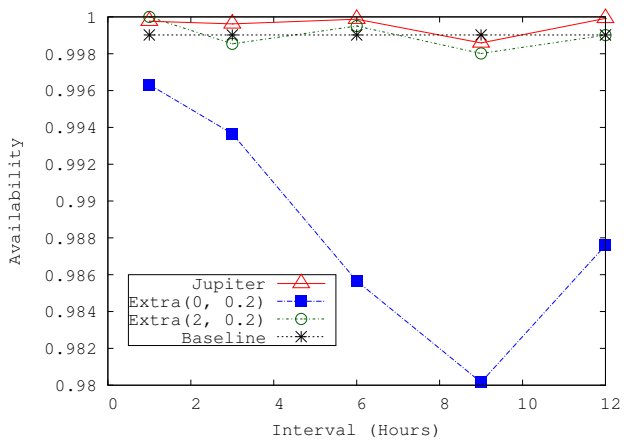

Figure 9: Availability of an erasure code based distributed storage service under different bidding strategies from October 2014 to December 2014

framework is a little higher than the strategy Extra(0, 0.2). The cost of strategy Extra(2, 0.2) is obviously higher than the other two bidding strategy.

There is no failure in the distributed lock service for our bidding framework besides the bidding interval of 12 hours (the availability is very close to 1.) as shown in Figure 7. The other two strategies cannot keep the same availability level at all. In the case of $Extra(0, 0.2)$, there are about 8 hours failure time in 11 weeks. This is far from the requirements of high availability. The availability of the distributed lock service under $Extra(2, 0.2)$ is higher than those under $Extra(0, 0.2)$. But it cannot always satisfy the constraints of service availability level in different bidding intervals.

For the $Extra(0, 0.2)$ strategy, the number of spot instances to bid is always 5. And $Extra(2, 0.2)$ strategy always bids 7 spot instances in each bidding interval. $Extra(2, 0.2)$ outperformed $Extra(0, 0.2)$ in the term of service availability level in all cases. This result demonstrates that system availability level is hard to keep without the failure probability of spot instances although the availability of distributed services can be improved with additional spot instances. Furthermore, $Extra(2, 0.2)$ was charged about \$ 31 $\sim$ \$ 41 extra cost for the two additional spot instances. Our bidding framework outperformed the $Extra(2, 0.2)$ strategy in both cost and availability.

For the distributed storage service, Figure 8 shows that the cost of the distributed storage service under our bidding framework varies in different bidding interval. The best case is still the bidding interval of 6 hours. The cost of distributed storage service in this case is \$ 189.93. The cost of distributed storage service with on-demand instances is reduced by more than \$ 1000. The cost under the strategy $Extra(0, 0.2)$ is about \$ 183.5 in average. It is a littler lower than the cost under our bidding framework. This is mainly due to the too much out-of-bid failures of spot instances under $Extra(0, 0.2)$.

Our bidding framework kept the service availability level of the baseline except for the case of bidding interval of 9 hours. In this case, the failure time of the distributed storage service is a little longer than the requirements of the

service availability level. Compared to our bidding framework, $Extra(0, 0.2)$ has a little lower cost but an unacceptable service availability level, and $Extra(2, 0.2)$ has a closed availability but much higher cost as shown in Figure 6 and 9.

In both Figure 6 and 8, the cost of our bidding framework changes a lot with different bidding intervals. A short bidding interval means launching new instances more frequently, which costs a lot on startup time. However, a long bidding interval loses some chance to change bidding with the spot prices. Our bidding framework should make higher bids for a longer bidding interval under availability consideration. The 6 hours bidding interval seems to be the most appropriate of all. An extension for our bidding framework is to detect the frequency of spot prices fluctuating and change the bidding interval correspondingly.

The "extra" strategies are simple and fixed. Without the knowledge of spot prices history data in each availability zone, making a same extra portion of bid is senseless. Furthermore, these strategies cannot keep high availability level as they have no spot instance failure probability estimation. Additional nodes and extra portion of bids reduce the failures potentially, but increase the cost. The results show that if using intuitive approaches, we cannot achieve both high availability and cost efficiency at the same time.

6. RELATED WORK

The advent of Spot Instance Market [30] has bring a new vision of cloud computing that computing instances can be traded like in a market where the price changes dynamically depending on the demand and supply. Research communities have shown a great interest on utilizing the spot instances for cost-efficient computing. Most of the current works focus on using spot instances for batch processing jobs such as MapReduce jobs. On the other hand, spot price models are proposed as the fluctuation of spot prices is fundamentally faced in the design of scheduling algorithms and fault tolerant mechanisms to be built over spot instance.

Chohan et al. [15] demonstrated that the execution of MapReduce jobs could speed up significantly by using spot instances as accelerators with an acceptable mount of monetary cost. Their work focused on bidding for performance using spot instances. There is no worry about the fault tolerance of computing jobs because the failures of spot instance have no impact on the execution of MapReduce jobs. The study of Chohan et al. revealed the possibility of using spot instances to speed up batching processing jobs. Instead of bidding for performance, our study mainly focuses on bidding for availability which is significantly different.

For MapReduce jobs, the failure of a task will not hurt the progress of the whole job. The tasks in such a job can be divided and scheduled to any available spot instance. As the bookkeeping task can run continuously by using normal on-demand instance during the lifetime of a job, all the tasks in the job will be scheduled and finished eventually. However, many MapReduce implementations, such as Hadoop [20], are not designed for the spot market environment. Even if the bookkeeping task does not run on a spot instance, several simultaneous spot instance terminations could cause all replicas of a data item to be lost. For fully taking advantage of spot instances, Spot Cloud MapReduce [24] was proposed as a MapReduce implementation that can make computing progress even if lots of nodes are in failure simultaneously. By using the spot instance, the total cost of MapReduce job can be reduced while the completion time might be a little longer.

For compute-intensive, embarrassingly parallel jobs, adaptive checkpointing and work migration schemes were introduced to eliminate the impact of unexpected job terminations in [36, 35]. Checkpointing and work migration are two commonly used fault tolerance techniques for parallel computing jobs. These studies retrofitted these two fault tolerance mechanisms for reducing the job completion time with failure-prone spot instances. As the failures of such computing jobs can be fixed by re-execution like schemes, the fluctuation of spot prices do not need to be considered in these studies.

However, all these studies are targeted on the divisible computing jobs, which can shift the time of processing to when the computing resources are available. This is impossible for distributed services, which should be as available as possible and cannot delay users' request arbitrary. Moreover, as these works address the issue of frequent spot instance failures under a fixed bid, efficient bidding strategies based on statistical analysis of the spot price history are not considered.

Andrzejak et al. used a probabilistic model to capture the relations of continuous changing spot prices and job termination probabilities in [3], then a pre-computed and fixed optimum bid for a computing task can be given under SLA constraints. This simple approach is not suitable for the case of frequent fluctuation of spot prices. From a cloud service broker's perspective, Song et al. proposed a profit aware dynamic bidding algorithm based on Lyapunov optimization technique in [31]. These studies made statistical analysis of the spot prices, but do not involve the availability analysis of distributed services.

The design of bidding strategies and spot instance failure model rely on the spot price model. Although Amazon EC2 does not disclose the details of its underlying pricing algorithm, the spot price model has been studied from outside Amazon EC2. A statistical analysis for all spot instances in Amazon EC2 was provided in [22]. Ben-Yehuda et al. conjectured that the spot prices were usually not market-driven but determined by a pricing algorithm based on auto-regressive model in Amazon EC2 before October 2011 in [1]. The Markovian property of the spot price sequences had been verified in [15, 31], A discrete semi-Markovian chain was further applied to model the spot price variations in [31]. In our work, we embedded the semi-Markovian chain into the spot instance failure model and discretized the sojourn time of the semi-Markovian Chain into minutes as suggested by the spot prices data nowadays.

7. CONCLUSIONS

This paper addressed the problem of bidding for availability when building distributed service with the spot instances offered by Amazon EC2. We have pointed out several challenges of keeping service avilability level of a distributed system when using spot instances. The analysis is complicated because of the spot instance failure model. The out-of-bid failure is the main failure of spot instances. This is different from the node failure in traditional distributed systems. For estimating the failure probability of spot instances, we employed a semi-Markovian chain to model the fluctuation of spot prices. The availability of a distributed

service built with spot instances can thus be estimated from the failure probabilities of instances instead of the number of simultaneous node failure that can be tolerant. The problem of bidding for availability is formalized as a non-linear programming model. The objective is to minimize the cost of spot instances and the constraint represents the availability requirements of the distributed service. However, solving this non-linear programming is NP-hard. Exhaustive search like methods is not practical at all. We presented a bidding framework to make practical solutions with a near-optimal bidding algorithm using enumerate and greedy strategy. Two fundamental distributed services, distributed lock service and erasure code based distributed storage service, are used to verify the effectiveness of the bidding framework. Our bidding framework can reduce the costs by 81.23% and 85.32% for lock service and storage service respectively while still keeping the availability level the same as it is by using on-demand instances.

8. ACKNOWLEDGMENTS

We would like to thank all the reviewers for their valuable comments and helpful suggestions. We also thank Yong Hu, Mingxing Zhang, Ce Guo for useful discussions about the spot price model and bidding framework. This work is supported by Natural Science Foundation of China (61433008, 61373145, 61170210, U1435216), National High-Tech R&D (863) Program of China (2012AA012600), Chinese Special Project of Science and Technology (2013zx01039-002-002).

9. REFERENCES

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing amazon ec2 spot instance pricing. *ACM Trans. Econ. Comput.*, 1(3):16:1–16:20, Sept. 2013.

[2] Y. Amir and A. Wool. Optimal availability quorum systems: Theory and practice. *Information Processing Letters*, 65(5):223 – 228, 1998.

[3] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under sla constraints. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '10, pages 257–266, Washington, DC, USA, 2010. IEEE Computer Society.

[4] `https://aws.amazon.com/about-aws/globalinfrastructure/`.

[5] `http://aws.amazon.com/ec2/instance-types/`.

[6] `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-limits.html`.

[7] `http://aws.amazon.com/ec2/sla/`.

[8] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[9] V. Barbu and N. Limnios. *Semi-Markov Chains and Hidden Semi-Markov Models Toward Applications: Their Use in Reliability and DNA Analysis*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[10] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 141–154, Berkeley, CA, USA, 2011. USENIX Association.

[11] `https://github.com/boto/boto`.

[12] D. R. Brillinger, P. M. Guttorp, and F. P. Schoenberg. Point processes, temporal. In *Encyclopedia of Environmetrics*. John Wiley & Sons, Ltd, 2006.

[13] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[14] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[15] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See spot run: using spot instances for mapreduce workflows. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 7–7. USENIX Association, 2010.

[16] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty- rst ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[18] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[19] G. Grimmett and D. Stirzaker. *Probability and random processes*, volume 2. Oxford Univ Press, 1992.

[20] `http://hadoop.apache.org`.

[21] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[22] B. Javadi, R. K. Thulasiramy, and R. Buyya. Statistical modeling of spot instance prices in public

cloud environments. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 219–228, Washington, DC, USA, 2011. IEEE Computer Society.

[23] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[24] H. Liu. Cutting mapreduce cost with spot market. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[25] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430, June 2012.

[26] S. Mu, K. Chen, Y. Wu, and W. Zheng. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 61–72, New York, NY, USA, 2014. ACM.

[27] D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation*, 123(2):210 – 223, 1995.

[28] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Comput. Commun. Rev.*, 27(2):24–36, Apr. 1997.

[29] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 325–336. VLDB Endowment, 2013.

[30] `http://aws.amazon.com/ec2/purchasing-options/spot-instances/`.

[31] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *INFOCOM, 2012 Proceedings IEEE*, pages 190–198. IEEE, 2012.

[32] M. Spasojevic and P. Berman. Voting as the optimal static pessimistic scheme for managing replicated data. *Parallel and Distributed Systems, IEEE Transactions on*, 5(1):64–73, Jan 1994.

[33] Z. Tong and R. Kain. Vote assignments in weighted voting mechanisms. In *Reliable Distributed Systems, 1988. Proceedings., Seventh Symposium on*, pages 138–143, Oct 1988.

[34] S. Wee. Debunking real-time pricing in cloud computing. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 585–590, May 2011.

[35] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2012.

[36] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 236–243, July 2010.