A Reverse Auction Market for Cloud Resources

Joris Roovers, Kurt Vanmechelen, and Jan Broeckhove

University of Antwerp,
Department of Computer Science and Mathematics,
Middelheimlaan 1, 2020 Antwerp, Belgium
kurt.vanmechelen@ua.ac.be

Abstract. The proliferation of the Infrastructure-as-a-Service (IaaS) paradigm has introduced possibilities for trading computational resources on a scale that moves beyond the individual provider level. At present however, the adoption of open markets for trading IaaS resources has been largely unexplored. This paper investigates the design of such an open market. Our focus thereby lies on flexibility and the ability to model and integrate currently deployed pricing schemes of real-world providers instead of imposing new schemes. We discuss the issues encountered by the Continuous Double Auction (CDA) in this regard and introduce a Continuous Reverse Auction (CRA) that is paired with a novel bidding language based on tag and constraint sets.

Keywords: Markets, Bidding Language, Reverse Auction, Double Auction, IaaS, Cloud Computing.

1 Introduction

Infrastructure-as-a-Service (IaaS) providers materialize the cloud computing model by offering consumers access to (virtualized) hardware resources while charging them based on how these resources are used. As the IaaS market continues to grow and the number of IaaS players continues to increase (as predicted by Gartner [6]), possibilities arise for trading IaaS resources on a scale that moves beyond the individual provider level. Such trading is used in other commodities markets such as electricity markets, to efficiently balance supply and demand on relatively short timescales and foster competition.

An important desirable property of such an envisioned market is that it needs to be *open*, accommodating the policies of current real-world providers. Only when a market has a sufficiently low entry barrier that brings together many different consumers and providers, will the price of computing resources be subject to market discipline rather than being dictated by specific providers. Realizing this criterion is non-trivial, and at present, the possibilities for creating such open markets have been largely unexplored. The complexity of an IaaS good is significantly higher than that of many other commodities as IaaS resources are heterogeneous and multi-dimensional. As standardization of resources and APIs among IaaS providers is currently non-existent, defining the exact properties of

K. Vanmechelen, J. Altmann, and O.F. Rana (Eds.): GECON 2011, LNCS 7150, pp. 32–45, 2012.

the goods that are being traded is non-trivial, and imposing a common standard is difficult. Additionally, many providers use different allocation and price models (all based on the pay-what-you-use model, but implemented differently) to charge their customers. Bringing together these different models into a single IaaS market is not addressed in current related work [3,5,11,7,2,1] on market mechanisms for utility and grid computing systems, and forms the focus of this paper.

In this contribution we investigate whether it is feasible to design such an open market for IaaS resources. We spend particular attention to the Continuous Double Auction (CDA) mechanism that is used in many commodity markets, and find it unsuitable for this purpose in the current IaaS setting due to the complexity and diversity of price models used by providers. We therefore introduce a new market mechanism called the *Continuous Reverse Auction* (CRA) that is combined with a flexible bidding language based on *tag* and *constraint* sets to deal with these shortcomings.

2 Tag and Constraint Sets

In order to introduce flexibility into the market with respect to heterogeneity of IaaS goods and pricing models, we rely on the use of *tag* and *constraint* sets. We extend the notion *resource sets* as defined by Dubé within the context of a CDA-based market for Grid resources [4], and accommodate it to the IaaS setting. Dubé introduces *resource sets* that describe resource properties in bids and offers to a CDA. Consider for example the resource set S in 1 which represents a computer with a x86 system architecture, two-core processor clocked at 2.4 GHz, 4GB of RAM and 16GB of hard disk space.

$$S = \{\mathrm{x86}_{\mathrm{proc\_arch}}, 2_{\mathrm{proc\_cores}}, 2.4_{\mathrm{proc\_clock}}, 4_{\mathrm{mem\_size}}, 16_{\mathrm{disk\_size}}\} \quad (1)$$

Obviously, the used unit as well as the type and domain of each of the items in the set should be clearly defined. In [4], this is done by specifying a discrete, finite domain set for each of the resource types, along with the used unit (if applicable). Examples of such domain sets are shown in 2.

$$\begin{aligned} \Phi_{\mathrm{proc\_cores}} &= \{1, 2, 4, 6, 8, 16, 32\} \\ \Phi_{\mathrm{mem\_size}} &= \{1, 2, 4, 6, 8, 16, 32, 64, 128, 256\}\ (\mathrm{GB}) \\ \Phi_{\mathrm{net\_bw}} &= \{10, 100, 1000, 2000\}\ (\mathrm{MB/s}) \end{aligned} \quad (2)$$

Given a *requirement* set R_b accompanying an ask and a *component* set C_o included in an offer:

$$\mathcal{R}_b \text{ matches } \mathcal{C}_o \iff \mathcal{C}_o \text{ matches } \mathcal{R}_b \iff \mathcal{R}_b \cap \mathcal{C}_o = \mathcal{R}_b \quad (3)$$

A shortcoming of this approach is that it lacks the expressiveness and flexibility that is needed to express some of the more complex constraints that are inherently part of IaaS solutions. For example, a constraint on *refund policy*

will typically include very provider specific conditions under which a refund can occur, as well as what this refund actually includes. As such, it cannot be expressed by a match on a single value. Another example is the implementation of a *region* constraint for which geographical knowledge is needed to determine which regions are included or overlap with others. A final issue is that in Dubé's approach a market is needed that has specific knowledge about each of the resource types in order to match sets, i.e. it needs to know the different matching semantics of each of the resources. In a situation where more and more complex constraints are used with complex matching semantics, using such a system is no longer viable. Indeed, given the heterogeneity of IaaS solutions today, using a matching mechanism in which the market must be aware of all possible resources and constraints is not manageable nor future proof.

To address these issues, we introduce *tag sets* and *constraint sets*, which enable the specification of more complex constraints. In particular, the specification of a constraint on a resource that a consumer is requesting or a provider is offering will be separated from the specification of the values that characterize that resource. Splitting the values from the matching semantics enables the specification of complex constraints that span multiple resources. Our approach to matching also removes the need for the market to have knowledge of each type of item in the sets. When a consumer or provider describes a resource, requirement, service or feature it uses tag sets that describe its different characteristics. Formally, a tag $\mathcal{T}$ in a tag set $\mathcal{TS}$ is defined as the tuple in 4. It contains a name and a set of attributes that define the tag.

$$\mathcal{T} = [\mathcal{N}_{\mathcal{T}}, \mathcal{A}_{\mathcal{T}}] \tag{4}$$

The name $\mathcal{N}_{\mathcal{T}}$ of the tag can be any regular string, but has to be unique for each type of tag and within a tag set. The attributes $\mathcal{A}_{\mathcal{T}}$ associated with the tag (see 5) are a set of key-value pairs.

$$\begin{aligned} \mathcal{A}_{\mathcal{T}} &= \{(x, f(x))\} \\ f &: x \in string \mapsto value \end{aligned} \tag{5}$$

These attributes contain the values that further characterize a given tag. For example, in the case of a `Disk` tag ($\mathcal{N}_{\mathcal{T}}$="Disk"), the tag attributes will contain both the unit in which the hard disk size is specified as well as the size of the disk, i.e. `[Disk, {(unit, GB), (value, 160)}]`. Attribute values do not have to be singular, they can also be of a compound nature. That is, it is also possible that these values are lists or nested tags. For example, if multiple operating systems are supported by a provider, they can be listed as follows.

```
[SupportedOperatingSystems, {(list, {
  [OperatingSystem, {(name, Windows), (version, 7)}],
  [OperatingSystem, {(name, Ubuntu), (version, 11.04)}]}
)}]
```

Tag sets can therefore group multiple attributes under a single tag, allowing for a more hierarchical approach compared to resource sets. A less visible difference to [4] is that tag sets themselves only describe the characteristic of a request or offer. The matching semantics are encoded by *constraint sets* that accompany tag sets. These contain constraints that define a certain matching restriction. Formally, a constraint (6) is a function that defines a matching restriction on a request $\mathcal{R}$, by expressing restrictions on the tags of $\mathcal{R}$ and the tags of an offer $\mathcal{O}$ against which $\mathcal{R}$ is matched.

$$\mathcal{C} : (\mathcal{TS}_{\mathcal{R}}, \mathcal{TS}_{\mathcal{O}}) \mapsto \{true, false\} \tag{6}$$

Given the generic definition in 6, it is clear that the outcome of a constraint function can depend on a large number of factors. As an example, consider a memory constraint as given by Algorithm 1.

Algorithm 1. MemoryConstraint($\mathcal{TS}_1$, $\mathcal{TS}_2$)

```
T1 ← TS1["Memory"];
T2 ← TS2["Memory"];
if T1 ≠ null and T2 ≠ null then
  if (A_T1["unit"] = A_T2["unit"]) and (A_T1["value"] ≤ A_T2["value"]) then
    return true;
  end if
end if
return false;
```

A constraint function always takes two arguments: the tag set ($\mathcal{TS}_1$) that belongs to the request to which the constraint belongs and the tag set ($\mathcal{TS}_2$) associated with the offer against which the constraint is verified. The `Memory` constraint works as follows. First, the occurrence of the `Memory` tag is evaluated in both tag sets. If one of both sets does not contain the tag, the result of the constraint is negative. Otherwise, the "unit" attributes are compared first to verify whether both tags are specified in the same unit[1], and then checks whether "value" specified in $\mathcal{T}_1$ is less than or equal to that of $\mathcal{T}_2$. If this is the case, the constraint is fulfilled. In this example, the `Memory` constraint was used to define matching semantics. Besides the `Memory` constraint, many other constraints obviously exist. Some of these operations should be standardized by the market to avoid confusion among the market participants, while others can remain consumer- or provider-specific. An example where a consumer would specify a custom constraint is the previously mentioned case where a trade-off between resource uptime and refund policy is required.

Note that verifying consumer- or provider-specific constraints requires that the algorithm embodying the constraint is made available to the entity performing the matching. The most obvious way to do this, is by allowing consumers and

[1] The operation could be modified to contain logic to do unit conversions.

providers to run custom code that can verify constraints within the market during the matching process. Despite the fact that the action of running custom programming code on a remote location itself is relatively straightforward using a proper platform and software libraries, doing so introduces a set of technical challenges w.r.t. code verification and protecting the market's integrity.

By giving the market a database of standard constraints for which it knows the matching semantics and leaving the matching of non-standardized constraints to the consumer and provider themselves, the technical difficulties that remote code introduces are effectively avoided. In practice, such a system lets consumers and providers specify the name of the constraint as part of the constraint set when a standard constraint is used and specify the endpoint of a web service that can verify custom constraints in the other cases.

To increase the number of constraints that can be checked locally, we introduce a set of generic constraints such as the `LessThan` relational constraint defined in Algorithm 2. In order to express which tag is checked in a generic constraint, the notation `Constraint[Parameters...]` (as in `Resource[Memory]` will be used in the remainder of this contribution.

Algorithm 2. LessThanConstraint($\mathcal{TS}_1$, $\mathcal{TS}_2$, tagName, attributeName)

$\mathcal{T}_1 \leftarrow \mathcal{TS}_1$[tagName];
$\mathcal{T}_2 \leftarrow \mathcal{TS}_2$[tagName];
if $\mathcal{T}_1 \neq$ null **and** $\mathcal{T}_2 \neq$ null **then**
  if $\mathcal{A}_{\mathcal{T}_1}$[attributeName] $<$ $\mathcal{A}_{\mathcal{T}_2}$[attributeName] **then**
    return true;
  end if
end if
return false;

Algorithm 3. ResourceConstraint($\mathcal{TS}_1$, $\mathcal{TS}_2$, tagName)

$\mathcal{T}_1 \leftarrow \mathcal{TS}_1$[tagName];
$\mathcal{T}_2 \leftarrow \mathcal{TS}_2$[tagName];
if $\mathcal{T}_1 \neq$ null **and** $\mathcal{T}_2 \neq$ null **then**
  if ($\mathcal{A}_{\mathcal{T}_1}$["unit"] $=$ $\mathcal{A}_{\mathcal{T}_2}$["unit"]) **and** ($\mathcal{A}_{\mathcal{T}_1}$["value"] $\leq$ $\mathcal{A}_{\mathcal{T}_2}$["value"]) **then**
    return true;
  end if
end if
return false;

To further indicate the usefulness of such constraints, consider the `Resource` constraint in Algorithm 3 which is a generalization of the `Memory` constraint of Algorithm 1 that can also be used to implement many other hardware related constraints (such as `Disk` or `CpuClock`).

3 Combining the CDA with Tag and Constraint Sets

The aforementioned tag and constraint sets could be used in conjunction with a traditional CDA approach. A bid or ask is then represented by a 5-tuple:

```
[Volume, Price, ExpirationDate, TagSet, ConstraintSet]
```

As an example, consider the case where Amazon submits an ask to a CDA, specifying a volume of 5 small instances at the price of $0.10 per hour :

```
Ask = [5, 0.10, 12/07/11 13:00:00 UTC-7, {
        [Memory, {(unit, MB), (value, 1700)}], [EC2_ECU, {(value, 1)}],
        [Disk, {(unit, GB), (value, 160)}], [Platform, {(value, x86)}],
        [EC2_API, {(value, m1.small)}], [SLA_Uptime, {(value, 99.95)}],
        [SupportedOperatingSystems, {(list, ...)}],
        [SupportedEC2_AMI_IDS, {(list, ...)}],
        [Splittable, {(value, 1)}],
        [AllocationType, {(value, OnDemand)}],
        [Time, {(value, 1), (unit, hour)}],
        [Region, {(value, USA/Virginia)}]
        [Provider, {(name, Amazon AWS), (url, http://aws.amazon.com)}] },
      { ConsumerLocationExclusion["Cuba"], ... }
      ]
```

A consumer submits a bid with a price of $0.12 and some hardware constraints.

```
Bid = [2, 0.12, 05/06/11 02:00:00 UTC+1, {
         [Disk, {(unit, GB), (value, 100)}],
         [Memory, {(unit, MB), (value, 1500)}],
         [AllocationType, {(value, OnDemand)}],
         [Time, {(value, 1), (unit, hour)}],
         [Consumer, {(name, "Jan Wolk"),
         (location, "Brussels, Belgium, EU")}] },
       { Resource["Disk"], Resource["Memory"],
         Equal["AllocationType", "value"], EqualAll["Time"] }
       ]
```

Note the flexibility that our tags introduce with respect to describing the resources and conditions under which the good is traded. The `Splittable` tag for example allows the provider to indicate whether its offered volume can be partially matched or not and constrains the granularity of the split. Further note that a constraint is specified on the ask which constrains the geographical location of the consumer with the matching bid (e.g. to conform to trading agreements).

Although the use of tag and constraint sets increases the flexibility of the CDA, a fundamental problem remains that relates to the way real-world IaaS resources are currently priced. The price that consumers pay for a set of resources is not only determined by their defining characteristics, but also by *how* these resources are used. The total usage cost is not determined by a single price (e.g. cost per VM hour) but depends on a variety of different price components (e.g.

cost per GB inbound bandwidth consumed) that have different weights according to the actual resource usage. Because of this, adopting a CDA that only takes a single price component into account to sort the bid and ask queues is not desirable. If such a CDA would be used, it is destined to be gamed by providers that advertise very low prices for that component while charging high prices for the use of other resource components, thus leading to inefficient allocations.

Introducing a multi-price component model is possible if consumers add usage values to their tag sets, and providers provide component-wise prices in their offers. The market could then determine the transfer price for matching bid b with ask a as in 7.

$$P(a,b) = \sum_{\mathcal{T} \in \ (\mathcal{TS}_a \ \cup \ \mathcal{TS}_b)} v(\mathcal{T}) \ p_A(\mathcal{T}) \tag{7}$$

where the price $P(a,b)$ for a certain bid b that is matched with an ask a of provider A is determined by the weighted sum (according to the usage of the consumer) of the different price components specified by the provider's ask. The weights are given by $v(\mathcal{T})$, which is defined to be the value of the usage-tag that is used by the price-component for the resource specified by tag $\mathcal{T}$. The term $p_A(\mathcal{T})$ is then the unit price specified by provider A for usage of the resource specified by tag $\mathcal{T}$. A provider would need to encode the relation between usage tags and price tags (e.g. in a table) as shown in Table 1.

Table 1. Provider specified table that defines the relations between usage and price tags

Resource Tag	**Usage Tag**	**Usage unit**	**Unit price**
Memory CpuClock CpuCores Disk	TimeUsage	hour	0.10
OS (Windows based)	TimeUsage	hour	0.03
LoadBalancing	TimeUsage	hour	0.10
LoadBalancing	OutgoingBandwidthUsage	GB	0.02
LoadBalancing	IncomingBandwidthUsage	GB	0.02
OutgoingBandwidth	OutgoingBandwidthUsage	GB	0.13
Disk I/O	DiskAccessUsage	million accesses	0.05

By grouping the resources, providers can indicate that the consumer should only be charged once for those resources. Similarly, by adding a resource multiple times with different usage tags, the provider can indicate that certain resources incur costs that are determined by multiple factors (in the case of the load balancing service, both per hour and per in- or outgoing GB).

However, this still requires the transfer price to be a linear combination of the component prices, which is not the case in practice. Consider for example

NewServers [8] that provide 3GB of free outgoing bandwidth per hour, charging $0.10 per additional GB, or Amazon's tiered S3 pricing model.

4 Continuous Reverse Auction for IaaS Resources

The Continuous Reverse Auction (CRA) combines properties of the Reverse Auction and the Continuous Double Auction to trade IaaS resources among multiple buyers and sellers to deal with aforementioned issues. A systematic overview of how the CRA works, is provided in Figure 1.

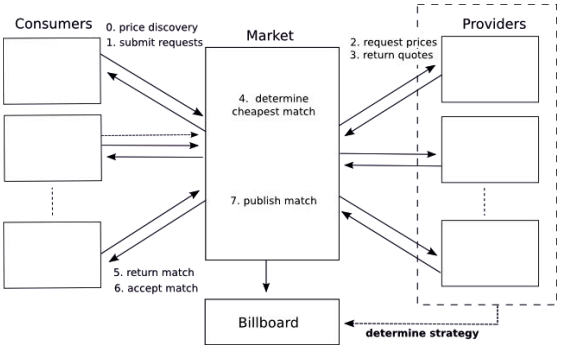

Fig. 1. Schematic overview of the workings of the Continuous Reverse Auction

4.1 Mechanism and Principles

In the Continuous Reverse Auction, consumers direct *requests* to the market, while providers deliver *quotes* in response to these requests. A consumers submits a requests to the market that subsequently returns the cheapest quote by querying a group of previously registered providers. Providers reply by either sending a quote to the market or by indicating that they are unable to match the given request. Upon receiving the quote from the market, the consumer can either accept or decline it.

The bidding process is initiated by a consumer that wants to allocate a set of IaaS resources and submits a request to the market. The bidding language that is used to specify this request, is based on the tag and constraint sets that were introduced in section 2:

```
Request = [Volume, TagSet, ConstraintSet]
```

A consumer can decide to accept or decline an offered quote, it is therefore no longer necessary to specify a price as part of his request. Also, as a request in the CRA is either matched immediately or not at all (if no provider is able to fulfill the consumer specified constraints), there is no need to specify an expiration date as part of the request. The components of a request are a `Volume` indicating

how many units a consumer wishes to allocate, and a `TagSet` and `Constraint` specifying the resource to allocate.

Once the request has arrived at the market, a price request is sent out to all registered providers, to which the providers respond with a quote or with a reply that they cannot match the request.

```
Quote = [Price, Volume, Identifier]
```

Besides the `Price`, the quote also contains a `Volume` that indicates how many units the provider can or is willing to match. The (securely encoded) `Identifier` in the quote allows a consumer to actually allocate resources after a match has been found on the market; it allows the consumer to prove the commitment a provider made when it handed out a certain quote. For further details we refer to [9].

After all registered providers have replied, the market returns the cheapest quote as a match to the consumer. If bids can be split, the market can match different units of a consumer's request with different providers and return multiple matches. Naturally, this matching occurs according to the quote prices specified by the providers (cheaper providers are matched first). Algorithm 4 outlines the matching procedure[2].

Algorithm 4. submitRequest(r: `Request`): Match

```
bestQuotes ← {}
for p in providers do
  bestQuote ← bestQuotes.first();
  currentQuote ← p.getQuote(r);
  if bestQuote = null or currentQuote.getPrice() < bestQuote.getPrice() then
    bestQuotes ← {currentQuote}
  else if bestQuote.getPrice() = currentQuote.getPrice() then
    bestQuotes.add(currentQuote)
  end if
end for
matchingQuote ← selectRandomElement(bestQuotes);
return new Match(matchingQuote);
```

As providers no longer submit asks to the market but are asked to deliver a quote on the basis of an individual request, providers are free to apply their own pricing scheme. Additionally, as providers now determine whether they can match a request with one of their offers or not, the market no longer needs to know how to match constraints, nor does it need to have constraint-specific knowledge. Indeed, when using the CRA mechanism, the market is constraint-agnostic, and acts only as an intermediate. Note that within the CRA, a consumer does not specify a maximum price as part of his request. This frees the

[2] In order to be concise, Algorithm 4 does not incorporate splittable bids.

consumer from the task of performing an accurate price estimation for a request. By allowing the consumer to accept or decline a quote (match) given its price, the consumer can do accurate price discovery and individual rationality is maintained.

4.2 Price Discovery

Price discovery is the process by which buyers and sellers determine or approximate the price of a good in the marketplace. Consumers in particular can make use of price discovery techniques to make an upfront trade-off between the price they will need to pay on the market and the resources and quality of service they will receive for that price. Additionally, price discovery techniques allow consumers and providers to determine bidding strategies by monitoring the price for a specific resource in the market and choosing the time at which they buy or sell certain resources intelligently as to maximize their personal profit.

In the CRA market, price discovery is in fact already built-in for consumers as they have the possibility to request a quote from the market at any given time without any further obligation. As such, both new as existing consumers can always obtain a quote for a request in order to direct their actions. Providers however, have no idea at which price and rate certain resources are being traded except when they are actually matched to a specific request.

In order to solve this lack of price discovery tools for providers, the CRA includes a billboard on which the market publishes anonymized accepted matches. This is done in real-time; after a match is found, it is directly published on the billboard. Because of this real-time behavior, the billboard effectively represents the current market situation. Providers can analyze the anonymized matches and adjust their pricing strategy accordingly. If a provider is able to infer that the price of matches in a specific category of requests is consistently lower than the price it is offering, it can adjust its strategy in order to get more matches. Analogously, it might increase the quote price for a category of requests, in order to increase its profit. As such indirect competition among providers is introduced and competitive prices can be formed. In future work we will focus on the development and experimental analysis of such strategies.

4.3 Sharing Semantics of Tag and Constraint Sets

In order to guarantee correct matching behavior, constraints should be precisely defined so that there is a clear and unique understanding among all market participants of what the constraint exactly specifies. As it is a core characteristic of the CRA that the market itself is unaware of constraint semantics, an additional entity is required to provide these. We therefore introduce a separate (market independent) *constraint catalogue* service that clearly specifies how constraint names map to actual matching semantics. In order to specify constraint semantics, the service can make use of a set of (algebraic) rules, pseudocode or a specific programming language.

As constraints are a core part of the contract between provider and consumer when a match is accepted, it is paramount that catalogue services are officially recognized and accredited by a mutually trusted third party,. This third party (which could potentially be the market itself), can then be trusted to objectively verify whether the contract is met by both parties or not. Such verification entails validating that the consumer has made correct payments, as well as verifying that provided resources actually meet the constraints specified in the consumer's request[3]. Note that the concept of the catalogue does not necessarily need to materialize into a web service; as long as there is a way for providers and consumers to share the semantics of particular constraints. In many cases it suffices that a provider states the meaning of a certain constraint on its website.

In order to lower the entry-barrier for providers, a standard set of common constraints can be provided as a software library. This way, providers will not need to download code from the constraint catalogue or implement the constraints themselves based on the catalogue's definition. If providers internally model their offers using tag and constraint sets, they can even use the software library to determine whether a given request matches one of their offers or not. Alternatively intermediaries such as brokers can perform this function. Additionally, providing such a library will limit the proliferation of constraints that have similar or the same semantics. A consumer specifying a constraint includes the qualified name of the class that implements the desired constraint as part of a `ConstraintDescription` that is attached to its request. A provider then creates an instance of the specified constraint class[4], and calls the `validate` method on the resulting object, passing the consumers tags, the tags used to model the provider's offer and the constraint parameters as specified by the consumer.

Although using constraints with well specified semantics is the main requirement for market participants, it is unlikely that many matches will occur if the tags that consumers and providers use to model resources, requirements, services and features are not standardized. That is, most (if not all) constraints require that there are tags with specific names or attributes in the tag set that accompanies the constraint. For example, when using the `ResourceConstraint`, a tag with a specific name that has `value` and `unit` attributes is required in both the consumer and the provider tag set in order for a match to occur. To address this, catalogues should also define tags (names and associated attributes) besides constraint semantics. As is the case with constraint names, a reverse DNS naming convention can be used to avoid name clashes. The constraint library should include this set of standard tags (and the default catalogue service should consequently also contain them).

Besides standardized constraints, consumers also have the possibility to specify custom constraints that allow them to express specific matching behavior that is not available as part of the standard library or in any of the catalogue services. To do this, the consumer or its broker deploys a web service that is able to check whether

[3] Doing so is certainly non-trivial as it involves thoroughly examining and monitoring the allocated resources.

[4] This can be done generically using *reflection*, e.g. in the Java programming language.

a given constraint is met, given a tag set that was specified by the consumer and a tag set that describes a provider's offer. Figure 2 outlines this approach.

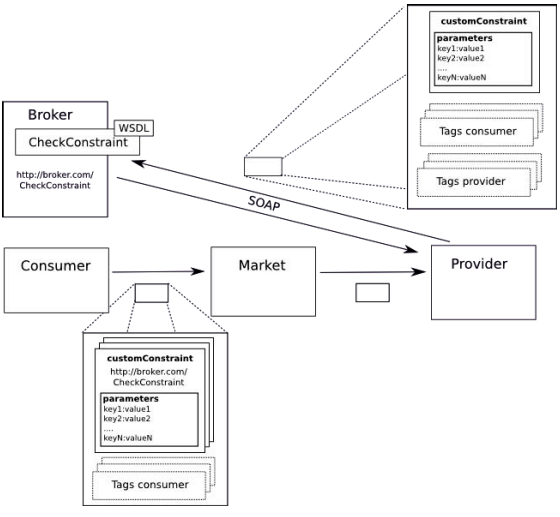

Fig. 2. Implementation of custom constraints using web services

Note that from an efficiency standpoint, it is preferable that a provider or the provider's broker first verifies all standardized constraints in order to decrease the probability that time-consuming web service calls need to be made.

A full implementation of our market based on Web Service technology (Java EE, JAX-WS, Spring, AJAX) is publicly available [10]. The implementation, which has been deployed to Amazon's EC2 platform, includes provider-side models for Amazon, GoGrid, CloudSigma and Rackspace, demonstrating that our bidding language can cope with the large amount of heterogeneity in resource specification and price models of current cloud providers.

4.4 Computational and Communicative Tractability

It is intuitively clear that the algorithmic complexity of the market's operation itself is rather low as the market's only task involves determining which of the quotes for a specific request contains the lowest price. By letting the providers determine whether they match a certain request as well as determine the price for that specific request, the market has effectively distributed the computational complexity that is inherent in matching and pricing heterogeneous IaaS resources. Furthermore, as no bids or asks need to be stored in queues as is the case in the double auction, no state (apart from the billboard) needs to be maintained between subsequent matches. As such, parallelizing and distributing the market's operation over different servers is possible, adding to attractiveness of the CRA mechanism from a computational point of view.

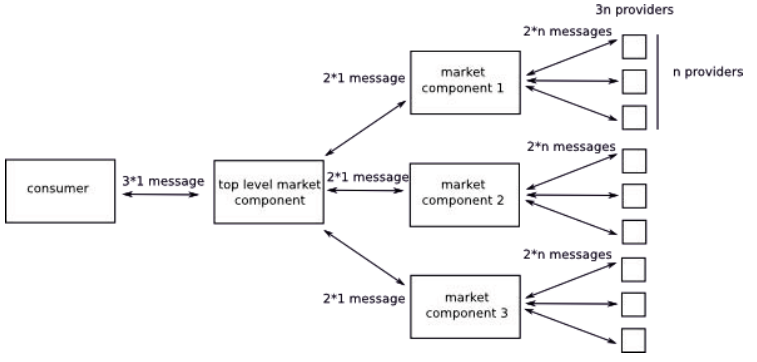

Fig. 3. Hierarchical market setup to reduce the potential for a communication bottleneck in the CRA

A downside of the CRA compared to a CDA is that the amount of communication between all involved market actors will be larger as the market sends out price requests to all providers for each consumer request. If there are n registered providers that can all fulfill a certain consumer request, $2n + 3$ messages will be needed for a match to occur. That is:

- 1 initial request from the consumer to the market
- n price requests from the market to the providers
- n responses from the providers to the market
- 1 match message from the market to the consumer
- 1 match acceptance message from the consumer to the market

As in the CDA no additional messages need to be exchanged with providers for an incoming consumer bid, it is clear that the flexibility the CRA offers comes at a considerable communication cost. In order to keep this cost manageable, providers might use brokers that have very good connectivity to the market. In order to prevent the network connectivity to the market to become a bottleneck, market process can be organized in a hierarchical manner, in which each node determines the cheapest match and subsequently forwards it to its parent component. An overview of this organization is shown in Figure 3.

5 Conclusion

Infrastructure as a Service providers are starting to embrace models that dynamically price their resources. Currently, the application of such models is constrained to individual providers and an open market for IaaS resources is yet to emerge. For such a market to materialize, flexible approaches are required that allow providers within the market to express their pricing schemes and resource models in line with their current modus operandi. The design of a market mechanism and bidding language that is flexible enough to realize this goal is an open research problem. In this contribution we discuss the issues encountered

by CDA-based approaches in this regard and introduce a Continuous Reverse Auction (CRA) that is paired with a novel bidding language based on tag and constraint sets. Our approach can accommodate the heterogeneity present in price and resource models currently used by providers, while allowing consumer bids to be matched with multiple providers. As such, it forms an important step towards a more structured and organized form IaaS resource trading.

References

1. Altmann, J., Courcoubetis, C., Stamoulis, G.D., Dramitinos, M., Rayna, T., Risch, M., Bannink, C.: GridEcon: A Market Place for Computing Resources. In: Altmann, J., Neumann, D., Fahringer, T. (eds.) GECON 2008. LNCS, vol. 5206, pp. 185–196. Springer, Heidelberg (2008)
2. Bapna, R., Das, S., Garfinkel, R., Stallaert, J.: A market design for grid computing. INFORMS Journal on Computing 20(1), 100–111 (2007)
3. Broberg, J., Venugopal, S., Buyya, R.: Market-oriented grids and utility computing: The state-of-the-art and future directions. Journal of Grid Computing 6(2), 255–276 (2008)
4. Dubé, N.: SuperComputing Futures: the Next Sharing Paradigm for HPC Resources. Ph.D. thesis, Université Laval (2008)
5. Fujiwara, I., Aida, K., Ono, I.: Applying double-sided combinational auctions to resource allocation in cloud computing. In: Proceedings of the 10th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT), pp. 7–14 (July 2010)
6. Leong, L.: Adopting cloud infrastructure as a service in the 'real world'. Tech. rep., Gartner (2011)
7. Neumann, D., Stoesser, J., Anandasivam, A., Borissov, N.: SORMA – Building an Open Grid Market for Grid Resource Allocation. In: Veit, D.J., Altmann, J. (eds.) GECON 2007. LNCS, vol. 4685, pp. 194–200. Springer, Heidelberg (2007)
8. NewServers: NewServers homepage (2009), `http://www.newservers.com/`
9. Roovers, J.: A Market Design for IaaS Cloud Resources. Master's thesis, University of Antwerp (2011), `http://thesisjorisroovers.blogspot.com`
10. Roovers, J., Vanmechelen, K.: IaaS Reverse Auction Market Prototype Implementation (2011), `http://jorisroovers.com/files/IaaSMarketPrototype.zip`
11. Vanmechelen, K., Depoorter, W., Broeckhove, J.: Combining futures and spot markets: A hybrid market approach to economic grid resource management. Journal of Grid Computing 9(1), 81–94 (2011)