Consumer Side Resource Accounting in the Cloud

Ahmed Mihoob, Carlos Molina–Jimenez and Santosh Shrivastava

School of Computing Science, Newcastle University
Newcastle upon Tyne, NE1 7RU, UK
{a.m.mihoob, carlos.molina, santosh.shrivastava}@ncl.ac.uk

Abstract. The paper investigates the possibility of consumer side resource accounting of cloud services, meaning, whether it is possible for a consumer to independently collect all the resource usage data required for calculating billing charges for pay-per-use cloud services. The accounting models of two widely used cloud services are examined and possible sources of difficulties are identified, including causes that could lead to discrepancies between the metering data collected by the consumer and the provider. The investigation is motivated by the fact that cloud service providers perform their own measurements to collect usage data, but as yet there are no equivalent facilities of consumer-trusted metering that are commonly provided by utility service providers like gas and electricity. The paper goes on to suggest how cloud service providers can improve their accounting models to facilitate the task of consumer side resource accounting.

Keywords: cloud resource consumption, storage and computational resources, resource metering and accounting models, Amazon Web Services

1 Introduction

Cloud computing services made available to consumers range from providing basic computational resources such as storage and compute power (infrastructure as a service, IaaS) to sophisticate enterprise application services (software as a service SaaS) A common business model is to charge consumers on a pay-per-use basis where they periodically pay for the resources they have consumed. Needless to say that for each pay-per-use service, consumers should be provided with an unambiguous resource accounting model that precisely describes all the constituent chargeable resources of the service and how billing charges are calculated from the resource usage (resource consumption) data collected on behalf of the consumer over a given period. If the consumers have access to such resource usage data then they can use it in many interesting ways, such as, making their applications billing aware, IT budget planning, create brokering services that automate the selection of services in line with user's needs and so forth. Indeed, it is in the interest of the service providers to make resource consumption data available to consumers; incidentally all the providers that we know of do make such data accessible to their consumers in a timely fashion.

An important issue that is raised is the *accountability* of the resource usage data: who performs the measurement to collect the resource usage data - the provider, the consumer, a trusted third party (TTP), or some combination of them[1]? The traditional utility providers such as water, gas and electricity perform their own measurements to collect usage data using metering devices (trusted by consumers) that are deployed in the consumers premises. Cloud service providers also perform their own measurements to collect usage data, although, as yet there are no equivalent facilities of consumer-trusted metering; rather, consumers have no choice but to take whatever usage data that is made available by the provider as trustworthy. A good introduction into the underlying trust issues can be found in [12].

In light of the above discussion, it is worth investigating whether it is possible for a consumer (or a TTP acting on behalf of the consumer) to independently collect all the resource usage data required for calculating billing charges. In effect, this means that a consumer (or a TTP) should be in a position to run their own metering service for measuring resource consumption. If this were possible, then consumers will be able to perform reasonableness checks on the resource usage data available from service providers as well as raise alarms when apparent discrepancies are suspected in consumption figures; furthermore, innovative charging schemes can be constructed with confidence by consumers who are themselves offering third party brokering services. In our earlier paper [11], we developed the notion of a *Consumer–centric Resource Accounting Model* for a cloud service. We say that a resource accounting model is *weakly consumer–centric* if all the data that the model requires for calculating billing charges can be queried programmatically from the provider. Further, we say that an accounting model is *strongly consumer–centric* if all the data that the model requires for calculating billing charges can be collected independently by the consumer (or a TTP). Strongly consumer–centric accounting models have the desirable property of openness and transparency, since service users are in a position to verify the charges billed to them. That paper also evaluated the accounting model of Simple Storage Service, S3 from Amazon to see how well it matches the proposed notion.

This paper contributes to the prior work in three ways: (i) the evaluation work on accounting models is extended to include a compute service (Amazon Elastic Compute Cloud, EC2) and we point out a few ambiguities in the EC2 model description (Section 3); (ii) we precisely identify the causes that could lead to discrepancies between the metering data collected by the provider and the consumer, and whether the discrepancies can be resolved (Section 4); and (iii) we present ideas on how an accounting model should be constructed so as to make them strongly consumer–centric (Section 5).

[1] A note on terminology: 'accountability' refers to concepts such as responsibility, answerability, trustworthiness; not to be confused with 'resource accounting' that refers to the process concerned with calculating financial charges.

2 Background

For resource accounting it is necessary to determine the amount of resources consumed by a given consumer (also called client and subscriber) during given time interval, for example, a billing period. **Accounting systems** are composed of three basic services: **metering**, **accounting** and **billing**.

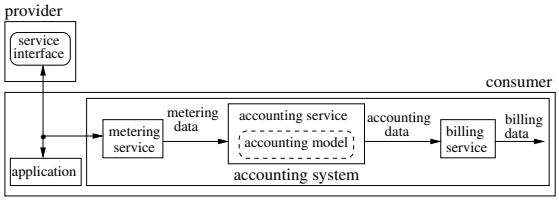

Fig. 1. Consumer side resource accounting system.

We show a consumer side accounting system in Fig.1. We assume that resources are exposed as services through one or more service interfaces. As shown in the figure, the metering service intercepts the message traffic between the consumer application and the cloud services and extracts relevant data required for calculating resource usage (for example, the message size which would be required for calculating bandwidth usage). The metering service stores the collected data for use by the accounting service. The accounting service retrieves the metering data, computes resource consumption from the data using its *accounting model* and generates accounting data that is needed by the billing service to calculate the billing data.

Accounting models are provider–specific in the sense that the functionality of an accounting model is determined by the provider's policies. These policies determine how the metrics produced by his metering service are to be interpreted; for example, 1.7 GB of storage consumption can be interpreted by the provider's accounting model either as 1 or 2 GB. The accounting models of cloud providers are normally available from their web pages and in principle can be used by subscriber to perform their own resource accounting. The difficulty here for the subscriber is to extract the accounting model from their online documentation as most providers that we know of, unnecessarily blur their accounting models with metering and billing parameters. The parameters involved in accounting models depend on the type of service (SaaS, PaaS, IaaS, etc.) offered. In this paper we will examine, from the point of view of consumer side resource accounting, the accounting models of Amazons Simple Storage Service (S3) and Elastic Compute Cloud (EC2). In the following discussion, we gloss over the fine details of pricing, but concentrate on metering and accounting services.

3 Accounting of Resource Consumption

3.1 S3 Accounting Model

An S3 space is organised as a collection of buckets which are similar to folders. A bucket can contain zero or more objects of up to 5 terabytes of data each. Both buckets and objects are identified by names (keys in Amazon terminology) chosen by the customer. S3 provides SOAP and RESTful interfaces. An S3 customer is charged for: a) **storage:** storage space consumed by the objects that they store in S3; b) **bandwidth:** network traffic generated by the operations that the customer executes against the S3 interface; and c) **operations:** number of operations that the customer executes against the S3 interface.

Storage: The key parameter in calculation of the storage bill is number of byte hours accounted to the customer. *Byte Hours* (ByteHrs) is the the number of bytes that a customer stores in their account for a given number of hours.

Amazon explains that *the GB of storage billed in a month is the average storage used throughout the month. This includes all object data and metadata stored in buckets that you created under your account. We measure your usage in TimedStorage–ByteHrs, which are added up at the end of the month to generate your monthly charges.* They further state that *at least twice a day, we check to see how much storage is used by all your Amazon S3 buckets. The result is multiplied by the amount of time passed since the last checkpoint.*

From the definition of ByteHrs it follows that to calculate their bill, a customer needs to understand 1) how their byte consumption is measured, that is, how the data and metadata that is uploaded is mapped into consumed bytes in S3; and 2) how Amazon determines the number of hours that a given piece of data was stored in S3 —this issue is directly related to the notion of a checkpoint.

Amazon explains that each object in S3 has, in addition to its data, system metadata and user metadata; furthermore it explains that the **system metadata** is generated and used by S3, whereas **user metadata** is defined and used only by the user and limited to 2 KB of size [1]. Unfortunately, Amazon does not explain how to calculate the actual storage space taken by data and metadata. To clarify the issue, we conducted a number of experiments (see [11]) involving uploading of a number of objects of different names, data and user metadata into an equal number of empty buckets.

Three conclusions can be drawn from these experiments: first, the mapping between bytes uploaded (as measured by intercepting upload requests) and bytes stored in S3 correspond one to one; second, the storage space occupied by system metadata is the sum of the lengths (in Bytes) of object and bucket names and incur storage consumption; third, user metadata does not impact storage consumption. In summary, for a given uploaded object, the consumer can accurately measure the total number of bytes that will be used for calculating ByteHrs.

Next, we need to measure the 'Hrs' of 'ByteHrs'. As stated earlier, Amazon states that at least twice a day they check the amount of storage consumed by a customer. However, Amazon does not stipulate exactly when the checkpoints take place.

To clarify the situation, we conducted a number of experiments that consisted in uploading to and deleting files from S3 and studying the Usage Reports of our account to detect when the impact of the PUT and DELETE operations were accounted by Amazon. Our findings are summarised in Fig.2. It seems that, currently, Amazon does not actually check customers' storage consumption twice a day as they specify in their Calculating Your Bill document, but only once. From our observations, it emerged that the time of the checkpoint is decided randomly by Amazon within the 00:00:00Z and 23:59:59Z time interval[2].

In the figure, CP stands for checkpoint, thus $CP_{30} : 2GB$ indicate that CP_{30} was conducted on the 30th day of the month at the time specified by the arrow and reported that at that time the customer had 2 GB stored in S3. SC stands for Storage Consumption and is explained below.

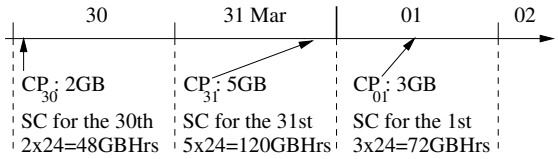

Fig. 2. Amazon's checkpoints.

As shown in the figure, Amazon uses the results produced by a checkpoint of a given day, to account the customer for the 24 hrs of that day, regardless of the operations that the customer might perform during the time left between the checkpoint and the 23:59:59Z hours of the day. For example, the storage consumption for the 30th will be taken as $2 \times 24 = 48$ GBHrs; where 2 represents the 2GB that the customer uploaded on the 30th and 24 represents the 24 hrs of the day.

Bandwidth: Amazon charges for the network data transferred from the customer to S3 ('DataTransfer–In', the request message of an operation) and the network data transferred from S3 to the customer (the corresponding response message, 'DataTransfer–Out'). It is however not clear from the available information how the size of of the message is calculated. We therefore conducted several experiments involving uploading, downloading, deleting etc. of objects using both RESTful and SOAP interfaces and compared the information extracted from the intercepted messages with the information available from Amazon usage reports. It turns out that for RESTful operations, only the size of the object (in DataTransfer–In for PUT, and DataTransfer–Out for GET) is taken into account and system and user metadata is not part of the overhead, whereas for SOAP operations, the total size of the message is taken into account.

[2] S3 servers are synchronised to the Universal Time Coordinated (UTC) which is also known as the Zulu Time (Z time) and in practice equivalent to the Greenwich Mean Time (GMT).

Operations: It is straightforward for a consumer to count the type and number of operations performed on S3. We note that an operation might fail to complete successfully. The error response in general contains information that helps identify the party responsible for the failure: the customer or the S3 infrastructure. For example, *NoSuckBucket* errors are caused by the customer when they try to upload a file into a non-existent bucket; whereas an *InternalError* code indicates that S3 is experiencing internal problems. Our understanding is that the consumer is charged for an operation, whether the operation succeeded or not.

3.2 EC2 Accounting Model

EC2 is a computation service offered by Amazon as an IaaS [4]. The service offers raw virtual CPUs (also called a Virtual Machine or VM) to subscribers. A subscriber is granted administrative privileges over his VM, that he can exercice by means of sending remote comands to the Amazon Cloud from his desktop computer. For example, he is expected to configure, launch, stop, re–launch, terminate, backup, etc. his VM. In return, the subscriber is free to choose the operating system (eg Windows or Linux) and applications to run. In EC2 terminology, a running virtual CPU is called an *instance* whereas the frozen bundle of software on disk that contains the libraries, applications and initial configuration settings that are used to launch an instance is called the *Amazon Machine Image (AMI)*.

Currently, Amazon offers six types of instances that differ from each other in four initial configuration parameters that cannot be changed at running time: amount of EC2 compute units that it delivers, size of their memory and local storage (also called ephemeral and instance storage) and the type of platform (32 or 64 bits). An EC2 compute unit is an Amazon unit and is defined as the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor. Thus Amazon offer small, large, extra large and other types of instances. For example, the default instance type is the *Small Instance* and is a 32 bit platform that delivers 1 EC2 compute unit and provided with 1.7 GB of memory and 160 GB of local storage. These types of instances are offered to subscribers under several billing models: **on–demand instances**, **reserved instances** and **spot instances**. In our discussion we will focus on on–demand instances.

Under the on–demand billing model, Amazon defines the unit of consumption of an instance as the *instance hour (instanceHrs)*. Currently, the cost of an instance hour of a small instance running Linux or Windows, is, respectively, 9.5 and 12 cents. On top of charges for instance hours, instance subscribers normally incur additional charges for data tranfer that the instances generates (Data Transfer In and Data Transfer Out) and for addtional infrastructure that the instance might need such as disk storage, IP addresses, monitoring facilities and others. As these additional charges are accounted and billed separately, we will leave them out of our discussion and focus only on instance hours charges.

The figures above imply that if a subscriber accrues 10 instanceHrs of a small instance consumption, running Linux, during a month, he will incur a charge of 95 cents at the end of the month.

In principle, the pricing tables publicly available from Amazon web pages should allow a subscriber to independently conduct his own accounting of EC2 consumption. In the absence of a well defined accounting model this is not a trivial exercise.

Insights into the EC2 accounting model are spread over several on–line documents from Amazon. Some insight into the definition of instance hour is provided in the *Amazon EC2 Pricing* document [3] (see just below the table of *On–demand Instances*) where it is stated that *Pricing is per instance–hour consumed for each instance, from the time an instance is launched until it is terminated. Each partial instance–hour consumed will be billed as a full hour.* This statement suggests that once an instance is launched it will incur at least an instance hours of consumption. For example, if the instance runs continuouly for 5 minutes, it will incur 1 instanceHrs; likewise, if the instance runs continuously for 90 minutes, it will incur 2 instanceHrs.

The problem with this definition is that it does not clarify when an instance is considered to be launched and terminated. Additional information about this issue is provided in the *Billing* section of FAQs [2], *Paying for What You Use* of the *Amazon Elastic Compute (Amazon EC2)* document [4] and in the *How You're Charged* section of the User Guide [5]. For example, in [4] it is stated that *Each instance will store its actual launch time. Thereafter, each instance will charge for its hours of execution at the beginning of each hour relative to the time it launched.*

From information extracted from the documents cited above it is clear that Amazon starts and stops counting instance hours as the instance is driven by the subscriber, through different states. Also, it is clear that Amazon instance hours are accrued from the execution of one or more individual sessions executed by the subscriber during the billing period. Within this context, a *session* starts and terminates when the subscriber launches and terminates, respectively, an instance.

Session–based accounting models for resources that involve several events and states that incur different consumptions, are conveniently described by Finite State Machines (FSMs). We will use a FSM to describe EC2 accounting model.

States of an instance session The states that an instance can reach during a session depend on the type of memory used by the AMI to store its boot (also called root) device. Currently, Amazon supports S3–backed and EBS–backed instances. EBS stands for Elastic Block Store and is a persistent storage that can be attached to an instance. The subscriber chooses between S3 or EBS backed instances at AMI creation time.

Unfortunatelly, the states that an instance can reach during a session are not well documented by Amazon. Yet after a careful examination of Amazon's online documentation we managed to build the FSM shown in Fig. 3–a).

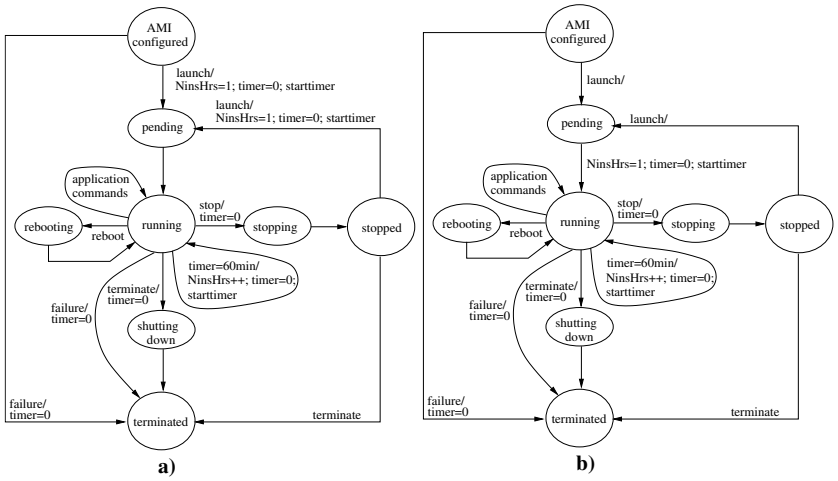

Fig. 3. Session of an Amazon instance represented as FSM.

The FSM of an Amazon instance includes two types of states: **permanent and transient states**. Permanent states (represented by large circles, e.g. *running*) can be remotely manipulated by commands issued by the subscriber; once the FSM reaches a permanent state, it remains there until the subscribers issues a command to force the FSM to progress to another state. Transient states (represented by small circles, e.g. *stopping*) are states that the FSM visits temporarily as it progresses from a permanent state into another. The subscriber has no control over the time spent in a transient state; this is why there are no labels on the outgoing arrows of these states.

We have labeled the transitions of the FSM with *event/action* notations. The *event* is the cause of the transition whereas the *action* represents the set (possibly empty) of operations that Amazon executes when the event occurs, to count the numbers of instance hours consumed by the instance.

There are two types of events: subscriber's and internal to the FSM events. The subscriber's events are the comands (*launch*, *application commands*, *reboot*, *stop* and *terminate*) that the subscribers issues to operate his instance; likewise, internal events are events that occur independently from the subscriber's commands, namely, $timer = 60min$ and *failure*.

AMI configured: is the initial state. It is reached when the subscriber sucessfully configures his AMI so that it is ready to be launched. **running:** is the state where the instance can perform useful computation for the subscriber, for example, it can respond to application commands issued by the subscriber. **terminated:** is the final state and represents the end of the life cycle of the instance. Once this state is reached the instance is destroyed. To perform additional computation after entering this state the subscriber needs to configure another AMI. The terminated state is reached when the subscribed issues the *terminate* command, the instance fails when it is in running state or the instance

fails to reach running state. **shuttingdown:** is reached when the subscriber issues the *terminate* command. **stopped:** this state is supported only EBS–backed instances (S3–backed instances cannot be stopped) and is reached when the user issues *stop* command, say for example, to perform backup duties. **rebooting:** is reached when the subscriber issues the *reboot* command.

States and and instance hours: In the figure, $NinstHrs$ is used to count the number of instance hours consumed by an instance during a single session. The number of instance hours consumed by an instance is determined by the integer value stored in $NinstHrs$ when the instance reaches the *terminated* state. *timer* is Amazon's timer to count 60 minutes intervals; it can be set to zero ($timer = 0$) and started ($starttimer$).

In the FSM, the charging operations are executed as suggested by the Amazon's on line documentation. For example, in *Paying for What You Use* Section of [4], Amazon states that the beginning of an instance hour is relative to the launch time. Consequently, the FSM sets $NinstHrs = 1$ when the subscriber executes a launch command from the *AMI configured* state. At the same time, *timer* is set to zero and started. $NinstHrs = 1$ indicates that once a subscriber executes a launch command, he will incur at least one instance hour. If the subscriber leaves his instance in the *running* state for 60 minutes ($timer = 60min$) the FSM increments $NinstHrs$ by one, sets the timer to zero and starts it again. From *running* state the timer is set to zero when the subscriber decides to terminate his instance (*terminate* command) or when the instance fails (*failure* event). Although Amazon's documentation does not discuss it, we believe that the possibility of an instance not reaching the *running* state cannot be ignore, therefore we have included a transition from *pending* to *terminated* state; the FSM sets the timer to zero when this abnormal event occurs.

As explained in *Basics of Amazon ESB–Backed AMIs and Instances* and *How You're Charged* of [5], a running ESB–backed instance can be stopped by the subscriber by means of the *stop* command and drive it to the *stopped* state. As indicated by $timer = 0$ operation executed when the subscribed issues a *stop* command, an instance in *stopped* state incurs no instance hours. However, though it is not shown in the figure as this is a different issue, Amazon charges for EBS storage and other addtional services related to the stopped instance. The subscriber can drive an instance from the *stopped* to the *terminated* state. Alternatively he can re–launch his instance. In fact, the subscriber can launch, stop and launch his instance as many times as he needs to. However, as indicated by the $NinstHrs++$, $timer = 0$ and $starttimer$ operations over the arrow, every transition from *stopped* to *pending* state accrues an instance hours of consumption, irrespectively of the time elapsed between each pair of consecutive *launch* commands.

Experiments with Amazon instances: To verify that the accounting model described by the FSM of Fig. 3–a) matches Amazon's description, we (as subscribers) conducted a series of practical experiments. In particular, our aim was to verify how the number of instance hours is counted by Amazon.

The experiments involved 1) configuration of different AMIs; 2) launch of instances; 3) execution of remote commands to drive the instances through the different states shown in the FSM. For example, we configured AMIs, launched and run them for periods of different lengths and terminated them. Likewise, we launched instances and terminated them as soon as they reached the *running* state.

To calculate the number of instance hours consumed by the instances, we recorded the time of execution of the remote commands *launch*, *stop*, *terminate* and *reboot*, and the time of reaching both transient and permanent states. For comparison, we collected data (start and end time of an instance hour, and number of instance hours consumed) from Amazon EC2 usage report.

A comparison of data collected from our experiments against Amazon's data from their usage report reveals that currently, the beginning of an instance hour is not the execution time of the subscriber's *launch* command, as documented by Amazon, but the time when the instance reaches the *running* state. These findings imply that the current accounting model currently in use is the one described by the FSM of Fig. 3–b). As shown in the figure, the $NinstHrs$ is incremented when the instance reaches the *running* state.

4 Potential Causes of Discrepancies

4.1 Storage

Since, for the calculation of ByteHrs, the time of the checkpoint is decided randomly by Amazon within the 00:00:00Z and 23:59:59Z time interval, the time used at the consumer's side need not match that at the provider's side: a potential cause for discrepancy. This is illustrated with the help of Fig.4.

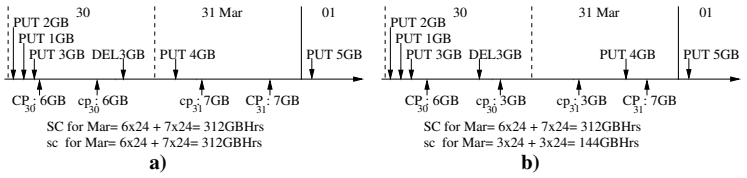

Fig. 4. Impact of checkpoints.

The figure shows the execution time of four PUT and one DEL operations executed by an S3 consumer during the last two days of March. The first day of April is also shown for completeness. For simplicity, the figure assumes that the earliest PUT operation is the very first executed by the consumer after opening his S3 account. The figure also shows the specific points in time when checkpoints are conducted independently by two parties, namely, Amazon and a consumer.

Thus, CP and cp represent, respectively, Amazon's and the consumer's checkpoints; the Giga Bytes shown next to CP and cp indicate the storage consumption detected by the checkpoint. For example, on the 30th, Amazon conducted its checkpoint about five in the morning and detected that, at that time, the customer had 6 GB stored ($CP_{30} : 6GB$). On the same day, the consumer conducted his checkpoint just after midday and detected that, at that time, he had 6 GB stored ($cp_{30} : 6GB$). SC and sc represent, respectively, the storage consumption for the month of March, calculated by Amazon and consumer, based on their checkpoints.

The figure demonstrates that the storage consumption calculated by Amazon and consumer might differ significantly depending on the number and nature of the operations conducted within the time interval determined by the two parties' checkpoints, for example, within CP_{31} and cp_{31}.

Scenario a) shows an ideal situation where no consumer's operations are executed within the pair of checkpoints conducted on the 30th or 31st. The result is that both parties calculate equal storage consumptions. In contrast, b) shows a worse–case scenario where the DEL operation is missed by CP_{30} and counted by cp_{30} and the PUT operation is missed by cp_{31} and counted by CP_{31}; the result of this is that Amazon and the consumer, calculate SC and sc, respectively, as 312 GB and 144 GB.

Ideally, Amazon's checkpoint times should be made known to consumers to prevent any such errors. Providing this information for upcoming checkpoints is perhaps not a sensible option for a storage provider, as the information could be 'misused' by a consumer by placing deletes and puts around the checkpoints in a manner that artificially reduces the consumption figures. An alternative would be to make the times of past checkpoints available (e.g., by releasing them the next day).

Impact of network and operation latencies: In the previous discussion concerning calculation of ByteHrs (illustrated using Fig. 4), we have implicitly assumed that the execution of a PUT (respectively a DELETE) operation is an atomic event whose time of occurrence is either less or greater than the checkpoint time (i.e., the operation happens either before or after the checkpoint). This allowed us to say that if the checkpoint time used at the provider is known to the consumer, then the consumer can match the ByteHrs figures of the provider. However, this assumption is over simplifying the distributed nature of the PUT (respectively a DELETE) operation. In Fig.5 we explicitly show network and operation execution latencies for a given operation, say PUT; also, *i, j, k and l* are provider side checkpoint times used for illustration. Assume that at the provider side, only the completed operations are taken into account for the calculation of ByteHrs; so a checkpoint taken at time i or j will not include the PUT operation (PUT has not yet completed), whereas a checkpoint taken a time k or l will. What happens at the consumer side will depend on which event (sending of the request or reception of the response) is taken to represent the occurrence of PUT. If the timestamp of the request message (PUT) is regarded as the time of occurrence of PUT, then the consumer side ByteHrs calculation

for a checkpoint at time i or j will include the PUT operation, a discrepancy since the provider did not! On the other hand, if the timestamp of the response message is regarded as the time of occurrence of PUT, then a checkpoint at time k will not include the PUT operation (whereas the provider has), again a discrepancy. In short, for the operations that occur 'sufficiently close' to the checkpoint time, there is no guarantee that they get ordered identically at both the sides with respect to the checkpoint time.

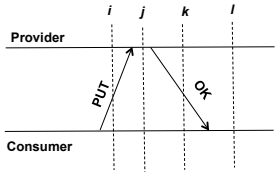

Fig. 5. Network and operation latencies.

Operations: Earlier we stated that it is straightforward for a consumer to count the type and number of operations performed on S3. There is a potential for discrepancy caused by network latency: operations that are invoked 'sufficiently close' to the end of an accounting period (say i) and counted by the consumer for that period, might get counted as performed in the next period (say j) by the provider if due to the latency, these invocation messages arrive in period j. This will lead to the accumulated charges for the two period not being the same. This is actually not an issue, as the Amazon uses the timestamp of the invocation message for resolution, so the consumer can match the provider's figure.

One likely source of difficulty about the charges for operations is determining the liable party for failed operations. Currently, this decision is taken unilaterally by Amazon. In this regard, we anticipate two potential sources of conflicts: DNS and propagation delays. As explained by Amazon, some requests might fail and produce a Temporary Redirect (HTTP code 307 error) due to temporary routing errors which are caused by the use of alternative DNS names and request redirection techniques [13]. Amazon's advice is to design applications that can handle redirect errors, for example, by resending a request after receiving a 307 code(see [1], Request Routing section). Strictly speaking these errors are not caused by the customer as the 307 code suggests. It is not clear to us who bears the cost of the re–tried operations.

4.2 EC2

The mismatch between Amazon's documented accounting model and the one currently in use (Fig. 3–a and b, respectively) might result in discrepances between the subscriber's and Amazon's calculations of instance hours. For example, imagine that it takes five minutes to reach the *running* state. Now imagine that

the subscriber launches an instance, leaves it running for 57 minutes and then terminates it. The subscriber's $NinstHours$ will be equal to two: $NinstHrs = 1$ at launch time and then $NinstHrs$ is incremented when $timer = 60min$. In contrast, to the subscriber's satisfaction, Amazon's usage records will show only one instance hour of consumption. One can argue that this discrepancy is not of the subscriber's concern since, economically, it always favours him.

More challenging and closer to the subscriber's concern are discrepancies caused by failures. Amazon's documentation does not stipulates how instances that fail accrue instance hours. For example, examine Fig. 3–a) and imagine that an instance suddenly crashes after spending 2 hrs and 15 min in *running* state. It is not clear to us whether Amazon will charge for the last 15 min of the execution as a whole instance hour. As a second example, imagine that after being launched either from *AMI configured* or *stopped* states, an instance progresses to *pending* state and from there, due to a failure, to *terminated*. It is not clear to us if Amazon will charge for the last instance hour counted by $NinstHrs$.

We believe that, apart from these omissions about failure situations, the accounting model of Fig. 3–a) can be implemented and used by the subscriber to produce accurate accounting. A salient feature of this model is that all the events (*launch*, *stop* and *terminate*) that impact the $NinstHrs$ counter are generated by subscriber. The only exeception if the $timer = 60min$ event, but that can be visible to the subscriber if he synchronises his clock to UTC time.

The accounting model that Amazon actually uses (Fig. 3–b) is not impacted by failures of instances to reach *running* state because in this model, $NinsHrs$ is incremented when the instance reaches *running* state. However, this model is harder for the subscriber to implement since the event that causes the instance to progress from *pending* to *running* state is not under the subscriber's control.

5 Related Work

An architecture for accounting and billing for resources consumed in a federated Grid infrastructure is suggested in [9]. The paper provides a valuable insight into the requirements (resource re–deployment, SLA awareness, pre–paid and post–paid billing, standardised records and others) that accounting and billing services should meet. In [6], the author discuss similar requirements for accounting and billing services, but within the context of federated network of telecommunication providers. Both papers overlook the need to provide consumers with means of performing consumer–side accounting. A detailed discussion of an accounting system similar to the one shown in Fig. 1 but aimed at telecommunication services is provided in [10].

Accounting models are fundamental to subscribers interested in planning for minimisation of expenditures on cloud resources. The questions raised are what workload to outsource, to which provider, what resources to rent, when, and so on. Examples of research results in this direction are reported in [14, 7]. In [7] the authors discuss how an accouting service deployed within an organisation

can be used to control expenditures on public cloud resources; their accounting service relies on data downloaded from the cloud provider instead of calculating it locally.

In [8], the authors take Amazon cloud as an example of cloud provider and estimate the performance and monetary–cost to compute a data–intensive (terabytes) workflow that requires hours of CPU time. The study is analytical (as opposite to experimental) and based on the authors' accouting model. For instance, to produce actual CPU–hours, they ignore the granularity of Amazon instance hours and assume CPU seconds of computation. This work stresses the relevance of accounting models.

6 Concluding Remarks

We investigated whether it is possible for a consumer (or a TTP acting on behalf of the consumer) to independently collect, for a given cloud service, all the resource usage data required for calculating billing charges. We examined two main IaaS services: storage and compute from Amazon; our investigation revealed the causes that could lead to discrepancies between the metering data collected by the the consumer not matching that of the provider. Essentially these causes can be classed into three categories discussed below.

1. Incompleteness and ambiguities: We pointed out several cases where an accounting model specification was ambiguous or not complete. For example, regarding bandwidth consumption, it is not clear from the available information what constitutes the size of of a message. It is only through experiments we worked out that for RESTful operations, only the size of the object is taken into account and system and user metadata is not part of the message size, whereas for SOAP operations, the total size of the message is taken into account. Failure handling is another area where there is lack of information and/or clarity: for example, concerning EC2, it is not clear how instances that fail accrue instance hours.
2. Unobservable events: If an accounting model uses one or more events that impact resource consumption, but these events are not observable to (or their occurrence cannot be deduced accurately by) the consumer, then the data collected at the consumer side could differ from the that of the provider. Calculation of storage consumption in S3 (ByteHrs) is a good example: here, the checkpoint event is not observable.
3. Differences in the measurement process: Difference can arise if the two sides use different techniques for data collection. Calculation of BytHrs again serves as a good example. We expect that for a checkpoint, the provider will directly measure the storage space actually occupied, whereas, for a given checkpoint time , the consumer will mimic the process by adding (for PUT) and subtracting (for DELETE) to calculate the space, and as we discussed with respect to Fig. 5, discrepancies are possible.

Issues raised by clauses 1 and 2 can be directly addressed by the providers. A provider should evaluate their accounting models by performing consumer

side accounting experiments to reveal any shortcommings. In particular, we recommend that for services that go through several state transitions (like EC2), providers should explicitly give FSM based descriptions, and ensure, as much as possible, that their models do not rely on unobservable (to consumer) events for billing charge calculations. On the whole, for IaaS services, consumer side accounting appears quite feasible. Whether this applies to PaaS and SaaS remains to be seen. Any discrepancies that get introduced unintentionally (e.g., due to non identical checkpoint times) can be resolved by consumers by careful examination of corresponding resource usage data from providers. Those that cannot be resolved would indicate errors on the side of consumers and/or providers leading to disputes.

References

1. Amazon: Amazon simple storage service. developer guide, API version 2006–03–01 (2006), wwww.amazon.com
2. Amazon: Amazon ec2 faqs (2011), aws.amazon.com/ec2/faqs
3. Amazon: Amazon ec2 pricing (2011), aws.amazon.com/ec2/pricing
4. Amazon: Amazon elastic compute cloud (amazon ec2) (2011), aws.amazon.com/EC2/
5. Amazon: Amazon elastic compute cloud user guide (api version 2011–02–28) (2011), docs.amazonwebservices.com/AWSEC2/latest/UserGuide/
6. Bhushan, B., Tschichholz, M., Leray, E., Donnelly, W.: Federated accounting: Service charging and billing in a business-to-business environment. In: Proc. 2001 IEEE/IFIP Int'l Symposium on Integrated Network Management VII. pp. 107–121 (2001)
7. den Bossche, R.V., Vanmechelen, K., Broeckhove, J.: Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads. In: Proc. IEEE 3rd Int'l Conf. on Cloud Computing(Cloud'10). pp. 228–235 (2010)
8. Deelman, E., Singh, G., Livny, M., Berriman, B., Good, J.: The cost of doing science on the cloud: The montage example. In: Proc. Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC'08) (2008)
9. Elmroth, E., Marquez, F.G., Henriksson, D., Ferrera, D.P.: Accounting and billing for federated cloud infrastructures. In: Proc. 8th Int'l Conf. on Grid and Cooperative Computing. pp. 268–275. Aug. 27–28, Lanzhou, Gansu, China (2009)
10. de Leastar, E., McGibney, J.: Flexible multi-service telecommunications accounting system. In: Proc. Int'l Network Conf. (INC'00) (2000)
11. Mihoob, A., Molina-Jimenez, C., Shrivastava, S.: A case for consumer–centric resource accounting models. In: Proc. IEEE 3rd Int'l Conf. on Cloud Computing(Cloud'10). pp. 506–512 (2010)
12. Molina-Jimenez, C., Cook, N., Shrivastava, S.: On the feasibility of bilaterally agreed accounting of resource consumption. In: 1st Int'l Workshop on Enabling Service Business Ecosystems (ESBE08). pp. 170–283. Sydney, Australia (2008)
13. Murty, J.: Programming Amazon Web Services. O'Reilly (2008)
14. Wang, H., Jing, Q., Chen, R., He, B., Qian, Z., Zhou, L.: Distributed systems meet economics: Pricing in the cloud. In: Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10) (2010)