



# COMPUTING SCIENCE

Consumer-centric Resource Accounting in the Cloud

Ahmed Mihoob, Carlos Molina–Jimenez and Santosh Shrivastava

**TECHNICAL REPORT SERIES**

---

No. CS-TR-1318

March 2012

## Consumer-centric Resource Accounting in the Cloud

A. Mihoob, C. Molina–Jimenez and S. Shrivastava

### Abstract

”Pay only for what you use” principle underpins the charging models of widely used cloud services that are on offer. An important issue then is the accountability of the resource usage data: who performs the measurement to collect resource usage data - the provider, the consumer, a trusted third party (TTP), or some combination of them? Provider-side accountability is the norm for the traditional utility services such as for water, gas and electricity, where providers make use of metering devices that are trusted by consumers. Currently, provider-side accountability is also the basis for cloud service providers, although, as yet there are no equivalent facilities of consumer-trusted metering; rather, consumers have no choice but to take whatever usage data made available by the provider as trustworthy. In light of this, the paper investigates whether it is possible for a consumer to independently collect all the resource usage data required for calculating billing charges for cloud services. 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. The paper proposes the notion of consumer-centric resource accounting model such that consumers can programmatically compute their consumption charges of a remotely used service. In particular, the notion of strongly consumercentric accounting model is proposed that requires that all the data needed 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 consumers are in a position to verify the charges billed to them. 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 paper goes on to suggest how cloud service providers can improve their accounting models to make them consumercentric.

## Bibliographical details

MIHOOB, A., MOLINA-JIMENEZ, C., SHRIVASTAVA. S.

Consumer-centric Resource Accounting in the Cloud  
[By] A. Mihoob, C. Molina-Jimenez, S. Shrivastava  
Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1318)

### Added entries

NEWCASTLE UNIVERSITY  
Computing Science. Technical Report Series. CS-TR-1318

### Abstract

"Pay only for what you use" principle underpins the charging models of widely used cloud services that are on offer. An important issue then is the accountability of the resource usage data: who performs the measurement to collect resource usage data - the provider, the consumer, a trusted third party (TTP), or some combination of them? Provider-side accountability is the norm for the traditional utility services such as for water, gas and electricity, where providers make use of metering devices that are trusted by consumers. Currently, provider-side accountability is also the basis for cloud service providers, although, as yet there are no equivalent facilities of consumer-trusted metering; rather, consumers have no choice but to take whatever usage data made available by the provider as trustworthy. In light of this, the paper investigates whether it is possible for a consumer to independently collect all the resource usage data required for calculating billing charges for cloud services. 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. The paper proposes the notion of consumer-centric resource accounting model such that consumers can programmatically compute their consumption charges of a remotely used service. In particular, the notion of strongly consumercentric accounting model is proposed that requires that all the data needed 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 consumers are in a position to verify the charges billed to them. 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 paper goes on to suggest how cloud service providers can improve their accounting models to make them consumercentric.

### About the authors

Ahmed Mihoob received his BSc in Computer Science from Sebha University- Libya in 1991, and his MSc in Computing Science from School of Computing Science - University of Newcastle in 2003. He is currently working towards his PhD under supervision of Prof. Santosh Shrivastava. His broad area of research interest is distributed computing, currently, his particularly interested in exploring pricing and billing of cloud services. Cloud computing allows consumers to perform computation in a public cloud with a pricing scheme typically based on incurred resource consumption. Consumers should be able to configure their applications to minimise charges. This in turn requires them to monitor and measure resource consumption to enable run time configuration management of applications. This is a relatively new research area that examines cloud computing from a new angle.

Carlos Molina-Jimenez received his PhD in the School of Computing Science at the University of Newcastle upon Tyne in 2000 for work on anonymous interactions in the Internet. He is currently a Research Associate in the School of Computing Science at the University of Newcastle upon Tyne where he is a member of the Distributed Systems Research Group. He is working on the EPSRC funded research project on Information Coordination and Sharing in Virtual Enterprises where he has been responsible for developing the Architectural Concepts of Virtual Organisations, Trust Management and Electronic Contracting.

Santosh Shrivastava was appointed a Professor of Computing Science, University of Newcastle upon Tyne in 1986. He received his Ph.D. in computer science from Cambridge University in 1975. Santosh's research interests are in the areas of computer networking, middleware and fault tolerant distributed computing. The emphasis of his work has been on the development of concepts, tools and techniques for constructing distributed fault-tolerant systems that make use of standard, commodity hardware and software components. His best known research work is the Arjuna distributed object transaction system (more than twelve years research effort, '85-98), funded by a succession of 5 EPSRC plus 4 EU, and 4 industry grants. This system led to a successful commercial product. You can find out more about it by visiting the pages of the distributed systems group. Other work includes OpenFlow transactional workflow management system, NewTop group communication system for atomic broadcasts and Voltan replicated processing system capable of tolerating Byzantine failures. The current focus of his work is on middleware for supporting inter-organization services where issues of trust, security, fault tolerance and ensuring

compliance to service contracts are of great importance as are the problems posed by scalability, service composition, orchestration and performance evaluation in highly dynamic settings. In 2005 Santosh received a platform grant from EPSRC for his research group to work on networked computing in inter-organisation settings. So far he has successfully supervised 23 PhD students (10 of whom are from abroad); many of these students are now in senior positions in industries and academia. He sits on programme committees of many international conferences/symposia; He is a member of IFIP WG6.11 on Electronic commerce - communication systems. He sits on the advisory board of Arjuna technologies Ltd.

### **Suggested keywords**

CLOUD RESOURCE CONSUMPTION  
STORAGE AND COMPUTATIONAL RESOURCES  
RESOURCE METERING AND ACCOUNTING MODELS  
AMAZON WEB SERVICES

# Consumer-centric Resource Accounting in the Cloud

Ahmed Mihoob  
School of Computing Science  
Newcastle University, UK  
a.m.mihoob@ncl.ac.uk

Carlos Molina-Jimenez  
School of Computing Science  
Newcastle University, UK  
carlos.molina@ncl.ac.uk

Santosh Shrivastava  
School of Computing Science  
Newcastle University, UK  
santosh.shrivastava@ncl.ac.uk

**Abstract**—“Pay only for what you use” principle underpins the charging models of widely used cloud services that are on offer. An important issue then is the accountability of the resource usage data: who performs the measurement to collect resource usage data - the provider, the consumer, a trusted third party (TTP), or some combination of them? Provider-side accountability is the norm for the traditional utility services such as for water, gas and electricity, where providers make use of metering devices that are trusted by consumers. Currently, provider-side accountability is also the basis for cloud service providers, although, as yet there are no equivalent facilities of consumer-trusted metering; rather, consumers have no choice but to take whatever usage data made available by the provider as trustworthy. In light of this, the paper investigates whether it is possible for a consumer to independently collect all the resource usage data required for calculating billing charges for cloud services. 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.

The paper proposes the notion of consumer-centric resource accounting model such that consumers can programmatically compute their consumption charges of a remotely used service. In particular, the notion of strongly consumer-centric accounting model is proposed that requires that all the data needed 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 consumers are in a position to verify the charges billed to them. 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 paper goes on to suggest how cloud service providers can improve their accounting models to make them consumer-centric.

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

## I. 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 sophisticated 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<sup>1</sup>? Provider-side accountability is the norm for the traditional utility services such as for water, gas and electricity, where providers make use of metering devices (trusted by consumers) that are deployed in the consumers’ premises. Currently, provider-side accountability is also the basis for cloud service providers, although, as yet there are no equivalent facilities of consumer-trusted metering; rather, consumers have no choice but to take whatever usage data made available by the provider as trustworthy.

In light of the above discussion, we propose the notion of a Consumer-centric Resource Accounting Model for a cloud resource. We say that an 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); in effect, this means that a consumer (or a TTP) should be in a position to run their own measurement service. We contend that it is in the interest of the providers to make the accounting models of their services at least weakly consumer-centric. Strongly consumer-centric models should prove even more attrac-

<sup>1</sup>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.

tive to consumers as they enable consumers to incorporate independent consistency/reasonableness checks as well as raise alarms when apparent discrepancies are suspected in consumption figures; furthermore, innovative charging schemes can be constructed by consumers that are themselves offering third party services. 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.

As a motivating example, consider a consumer who rents a storage service to run an application shown in Fig. 1. The storage is consumed by the consumer’s application and by applications hosted by other users ( $user_1$ ,  $user_2$ , etc.) that access the storage service at the consumer’s expense. An example of this case is a consumer using a storage service to provide photo or video sharing services to other users. The ideal scenario is that the consumer is able to instrument the application to collect all the necessary storage consumption data and use the accounting model of the provider to accurately estimate the charges, and use that information to provide competitively priced service to users.

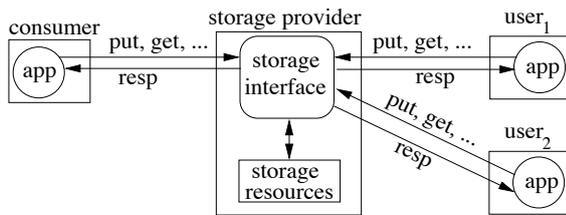


Figure 1. Provider, consumer and users of storage services.

Since cloud service providers do publish their charging information, it is worth investigating whether their information matches the proposed notion of consumer-centric resource accounting model. With this view in mind, we evaluated the accounting models of two cloud infrastructure services (simple storage service, S3, and Elastic Compute Cloud, EC2, both from Amazon) to see how well they match the proposed notion. We began by independently collecting (by examination of requests and responses) our own resource usage data for S3 and compared it with the provider’s data. Our investigations indicate that even though, it is conceptually a very simple service, the accounting model description of S3 nevertheless has a few ambiguities and not all the data that the model requires for calculating billing charges can be queried programmatically from the provider. A similar evaluation of EC2 also revealed a few ambiguities.

Learning from this exercise, 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. We present ideas on how an accounting model should be constructed so as to make them consumer-centric. We also suggest a systematic way of describing resource accounting models so that they can be understood and reasoned about by consumers.

Service providers can learn from our evaluation study to re-examine their accounting models. In particular, we recommend that a cloud provider should go through the exercise of constructing a third party measurement service, and based on that exercise, perform any amendments to the model, remove potential sources of ambiguities in the description of the model, so that as far as possible, consumers are able to collect with ease their own usage data that matches provider side data with sufficient precision<sup>2</sup>.

We begin by presenting the related work in this area; the following section (section three) presents the relevant background information on resource accounting. Section four examines the accounting models of S3 and EC2 from the point of view of consumer-centric resource accounting. Potential causes that could lead to discrepancies between resource consumption figures collected by providers and consumers are examined in section five. Section six presents the way forward, namely, how should resource accounting models be made consumer-centric and specified in a way that makes them easy to understand by consumers. Concluding remarks are presented in section seven.

## II. RELATED WORK

An architecture for accounting and billing for resources consumed in a federated Grid infrastructure is suggested in [3]. 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 [4], 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 aimed at telecommunication services is provided in [5].

An architecture for accounting and billing in cloud services composed out of two or more federated infrastructures (for example, a storage and computation providers) is discussed in [3]. The architecture assumes the existence of well defined accounting models that are used for accounting resources consumed by end users and for accounting resources that the cloud provider consumes from the composing infrastructures. This issue is related to the scenario that we present in Fig. 1.

In [6], the authors observe that “the black-box and dynamic nature of the cloud infrastructure” makes it difficult for consumers to “reason about the expenses that their applications incur”. The authors make a case for a framework for *verifiable resource accounting* such that a consumer can get assurances about two questions: (i) *did I* consume what I was charged? and (ii) *should I* have consumed what I was charged? Verifiability is clearly closely related to the notion of consumer-centric resource accounting developed in this paper.

<sup>2</sup>This paper combines and extends the material presented in two conference papers [1] and [2].

Our concept of consumer–centric resource accounting is similar in spirit to that of monitorability of service level agreements, discussed in [7]; in this work, the authors point out that service level agreements signed between clients and providers need to be precise and include only events that are visible to the client and other interested parties.

In [8], the authors develop a model in which the consumer and provider independently measure resource consumption, compare their outcomes and agree on a mutually trusted outcome. The paper discusses the technical issues that this matter involves, including consumer side collection of metering data, potential divergences between the two independently calculated bills, dispute resolution and non–repudiable sharing of resource usage records. Naturally, a starting point for such a system will be consumer–centric accounting models of cloud resources.

Good understanding of cloud resource accounting models is essential 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 [9], [10], [11]. In [10], the authors discuss how an accounting 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 [12], 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’ accounting 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. The suitability of Amazon S3, EC2 and SQS services as a platform for data intensive scientific applications is studied in [13]; the study focuses on performance (e.g. number of operations per second), availability and cost. It suggests that costs can be reduced by building cost–aware applications that exploit data usage patterns; for example, by favouring data derivation from raw data against storage of processed data. These arguments support the practical and commercial relevance of our study of resource accounting models.

### III. 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**.

We show a typical consumer side accounting system in Fig. 2. 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

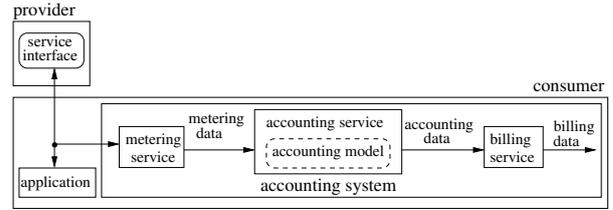


Figure 2. Consumer side resource accounting system.

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 a 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.

## IV. ACCOUNTING OF RESOURCE CONSUMPTION

### A. 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.

1) *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.* Next, an example that illustrates how to calculate your bill if you keep 2,684,354,560 bytes (or 2.5 GB) of data in your bucket for the entire month of March is provided. In accordance with Amazon the total number of bytes consumed for each day of March is 2684354560; thus the total number of ByteHrs is calculated as  $2684354560 \times 31 \times 24 = 1997159792640$ , which is equivalent to 2.5 GBMonths. At a price of 15 cents per Giga Bytes per month, the total charge amounts to  $2.5 \times 15 = 37.5$  cents.

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.* Records of storage consumption in ByteHrs can be retrieved from the Usage Reports associated with each account.

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 [14]. Unfortunately, Amazon does not explain how to calculate the actual storage space taken by data and metadata. To clarify this issue, we uploaded a number of objects of different names, data and user metadata into an equal number of empty buckets. Fig. 3 shows the parameters and results from one of our upload operations where an object named *Object.zip* is uploaded into a bucket named *MYBUCKET*, which was originally empty.

Notice that in this example, the object and bucket names are, respectively, ten and eight character long, which is equivalent to ten and eight bytes, respectively.

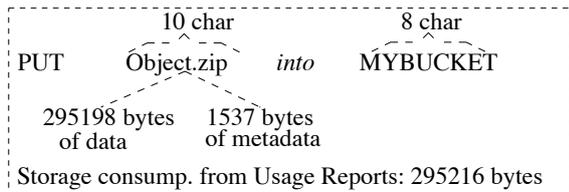


Figure 3. Impact of data and metadata on storage consumption.

The object data and metadata shown in the figure correspond to information we extracted locally from the PUT request. In contrast, the storage consumption of

295216 bytes corresponds to what we found in the Usage Reports. The actual Usage Reports show storage consumption per day in ByteHrs; the value shown is the result of its conversion into bytes. Notice that this storage consumption equals the sum of the object data, the length of the object name and the length of the bucket name:  $8 + 10 + 295198 = 295216$ .

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.4. 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<sup>3</sup>.

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.

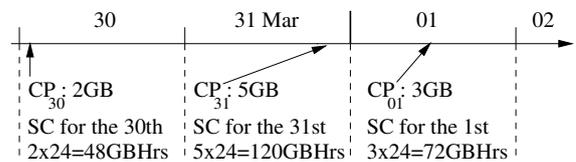


Figure 4. Amazon's checkpoints.

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

<sup>3</sup>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).

the 2GB that the customer uploaded on the 30th and 24 represents the 24 hrs of the day.

2) **Bandwidth:** Amazon explains that **DataTransfer-In** is the network data transferred from the customer to S3. They state that *Every time a request is received to put an object, the amount of network traffic involved in transmitting the object data, metadata, or keys is recorded here.* **DataTransfer-Out** is the network data transferred from S3 to the customer. They state that *Every time a request is received to get an object, the amount of network traffic involved in transmitting the object data, metadata, or keys is recorded here.* By here they mean that in the Usage Reports associated to each account, the amount of **DataTransfer-In** and **DataTransfer-Out** generated by a customer, is represented, respectively, by the **DataTransfer-In-Bytes** and **DataTransfer-Out-Bytes** parameters.

Amazon use an example to show that if *You upload one 500 MB file each day during the month of March and You download one 500 MB file each day during the month of March* your bill for March (imagine 2011) will be calculated as follows. The **DataTransfer-In** would be  $500MB \times (1/1024) \times 31 = 15.14GB$ . At a price of 10 cents per Giga Bytes, the total charge would be  $15.14 \times 10 = 151.4$  cents. In a second example they show that if *You download one 500 MB file each day during the month of March* the total amount of **DataTransfer-Out** would be 15.14 GB which charged at 15 cents per GB would amount to 227 cents.

It is however not clear from the available information how the size of of the message is calculated. To clarify the point, we uploaded a number of files and compared information extracted from the PUT operations against bandwidth consumption as counted in the Usage Report. Two examples of the experiments that we conducted are shown in Fig. 5: we used PUT operations to upload an object into a bucket. The data and metadata shown in the figure represent the data and metadata extracted locally from the PUT requests.

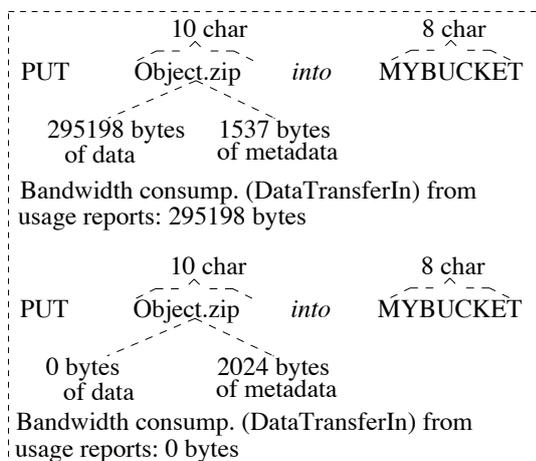


Figure 5. Bandwidth consumption.

As shown by the *Bandwidth consump.* parameters ex-

tracted from the Usage Reports, only the object data consumes **DataTransfer-In** bandwidth; neither the metadata or the object or bucket names seem to count as overhead. This observation refers to RESTful requests. In contrast, for SOAP messages, the total size of the message is always used for calculating bandwidth consumption

3) **Operations:** It is straightforward for a consumer to count the type and number of operations performed on S3. To illustrate their charging schema Amazon provide an example in the Amazon Simple Storage Service FAQs where You transfer 1000 files into Amazon S3 and transfer 2000 files out of Amazon S3 each day during the month of March, and delete 5000 files on March 31st. In this scenario, the total number of PUT request is calculated as  $1000 \cdot 31 = 31000$ , whereas the total number of GET requests is calculated as  $2000 \cdot 31 = 62000$ . The total number of DELETE requests is simply 5000 though this is irrelevant as DELETE requests are free. At the price of one cent per 1000 PUT requests and one cent per 10000 GET requests, the total charge for the operations is calculated as  $31000 (1/1000) + 62000 (1/10000) = 37.2$  cents.

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, *NoSuchBucket* 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.

To offer high availability, Amazon replicates data across multiple servers within its data centres. Replicas are kept weakly consistent and as a result, some perfectly legal operations could sometime fail or return inaccurate results (see [14], Data Consistency Model section). For example, the customer might receive a *ObjectDoesNotExist* as a response to a legal GET request or an incomplete list of objects after executing a LIST operation. Some of these problems can be corrected by re-trying the operation. From Amazon accounting model, it is not clear who bears the cost of the failed operations and their retries.

We executed a number of operations including both valid and invalid ones (for example, creation of buckets with invalid names and with names that already existed). Next we examined the Usage Reports and as we expected, we found that Amazon counted both successful and failed operations. Fig. 6 shows an example of the operations that we executed and the bandwidth and operation consumptions that it caused in accordance with the Usage Reports.

Thus, the failed operation to create that bucket consumed, respectively, 574 bytes and 514 bytes of **DataTransfer-In** and **DataTransfer-Out**. These figures, correspond to the size of the SOAP request and response, respectively. As shown in the figure, we also found out that the failed operation incurred operation consumption and counted by the **RequestTier2** parameter in the Usage Reports.

```

CREATE MYBUCKET // MYBUCKET already exists
Response: Error:BucketAlreadyExists
Bandwidth consump. (DataTransferIn) from
usage reports: 574 bytes
Bandwidth consump. (DataTransferOut) from
usage reports: 514 bytes
Operation consump. (RequestTier2) from
usage reports: 1

```

Figure 6. Bandwidth and operation consumption of failed operations.

## B. EC2 Accounting Model

EC2 is a computation service offered by Amazon as an IaaS [15]. The service offers raw virtual CPUs to subscribers. A subscriber is granted administrative privileges over his virtual CPU, that he can exercise by means of sending remote commands to the Amazon Cloud from his desktop computer. For example, he is expected to configure, launch, stop, re-launch, terminate, backup, etc. his virtual CPU. 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 a **Virtual Machine Instance (VMI)** or just 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 an **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* (*instanceHr*). Currently, the cost of an instance hour of a small instance running Linux or Windows, is, respectively, 8.5 and 12 cents. On top of charges for instance hours, instance subscribers normally incur additional charges for data transfer that the instances generates (**Data Transfer-In** and **Data Transfer-Out**) and for additional 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 85 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 [16] (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 continuously 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 [17], *Paying for What You Use* of the *Amazon Elastic Compute (Amazon EC2)* document [15] and in the *How You're Charged* section of the User Guide [18]. For example, in [15] 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.*

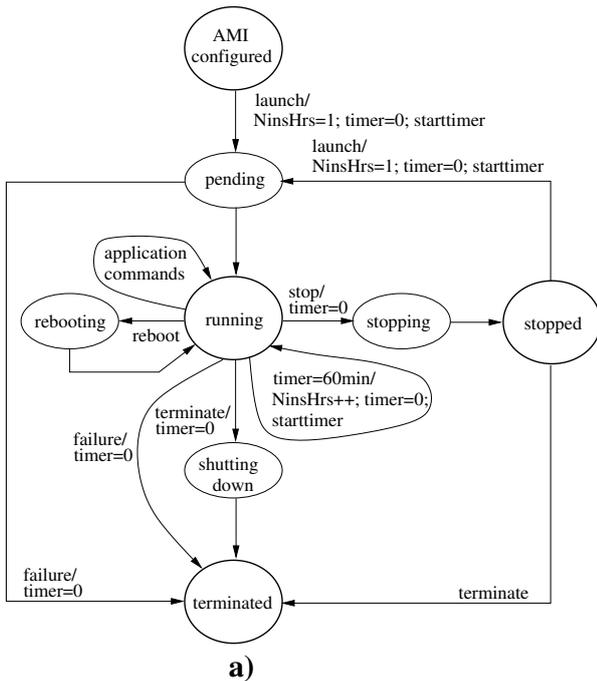
From information extracted from the documents cited above it is clear that Amazon starts and stops counting instance hours as the instance is driven by the subscriber, through different states. Also, it is clear that Amazon instance hours are accrued from the execution of one or more individual sessions 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.

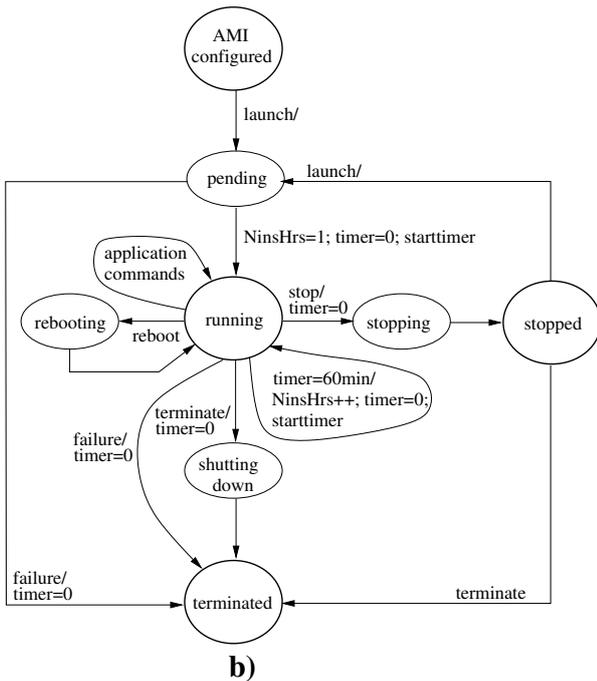
1) *States of an instance session*: The states that an instance can reach during a session depend on the type of memory used by the AMI to store its boot (also called root) device. Currently, Amazon supports S3-backed and EBS-backed instances. EBS stands for Elastic Block Store and is a persistent storage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

Unfortunately, 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 documen-

tation we managed to build the FSM shown in Fig. 7-a).



a)



b)

Figure 7. 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 commands (*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*. A discussion on all the permanent and some of the transient states depicted in the FSM follows.

- **AMI configured:** is the initial state. It is reached when the subscriber successfully 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.
- **Pending:** is related to the instantiation of the instance within the Amazon cloud. *Pending* leads to *running* state when the instance is successfully instantiated or to *terminated* state when Amazon fails to instantiate the instance.
- **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.

2) *States 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 a 60 minutes interval; 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 [15], 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 ignored, 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 EBS-Backed AMIs and Instances and How You’re Charged* of [18], a running EBS-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 subscriber 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 additional 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 hour of consumption, irrespectively of the time elapsed between each pair of consecutive *launch* commands.

3) *Experiments with Amazon instances*: To verify that the accounting model described by the FSM of Fig. 7-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. 7-b). As shown in the figure, the  $NinstHrs$  is incremented when the instance reaches the *running* state.

## V. POTENTIAL CAUSES OF DISCREPANCIES

### A. 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.8.

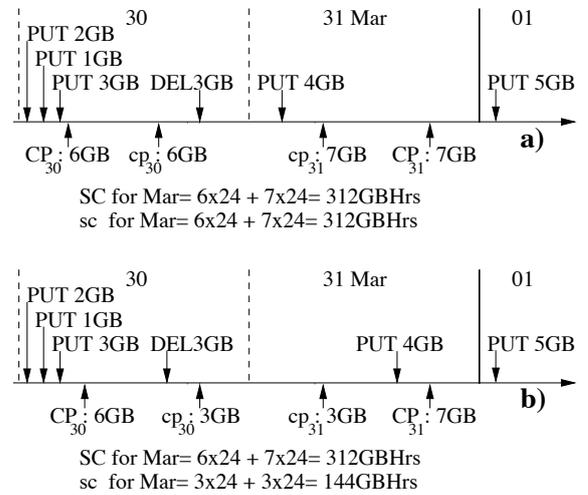


Figure 8. Impact of checkpoints.

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. 8), 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.9 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.

**Operations:** Earlier we stated that it is straightforward for a consumer to count the type and number of operations

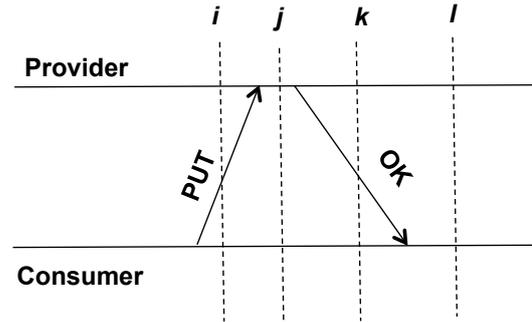


Figure 9. Network and operation latencies.

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 [19]. Amazon's advice is to design applications that can handle redirect errors, for example, by resending a request after receiving a 307 code (see [14], 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.

## B. EC2

The mismatch between Amazon's documented accounting model and the one currently in use (Fig. 7-a and b, respectively) might result in discrepancies 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:  $NinstHours = 1$  at launch time and then  $NinstHours$  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 stipulate how instances that fail accrue

instance hours. For example, examine Fig. 7–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. 7–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 exception is 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. 7–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.

## VI. CONSUMER-CENTRIC MODELS

Based on the understanding gained from our evaluation study, we suggest a systematic way of constructing and specifying consumer-centric resource accounting models. As observed earlier, 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. Our investigations revealed the causes that could lead to discrepancies between the metering data collected by the consumer not matching that of the provider. Essentially these causes can be classed into three categories discussed below.

- 1) Incompleteness and ambiguities: It is of course necessary that consumers are provided with an unambiguous resource accounting model that precisely describes all the constituent chargeable resources of a service and how billing charges are calculated from the resource usage (resource consumption) data collected on behalf of the consumer over a given period. 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 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. 9, discrepancies are possible.

Issues raised by clauses 1 and 2 can be directly addressed by the providers. In particular, we recommend that providers should evaluate their accounting models by performing consumer side accounting experiments to reveal any shortcomings. Further, 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. 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.

### A. Abstract resource

We suggest a systematic way of describing resource accounting models so that they can be understood and reasoned about by consumers. The key idea is very simple: first define a set of “elementary” chargeable resources and then describe the overall resource consumption of a given resource/service in terms of an aggregation of the consumption of these elementary resources. With this view in mind, we present the resource consumption model of an *abstract resource*. Next we will argue that with some small resource specific variations, the accounting models of resources such as S3, EC2, EBS and other infrastructure level resources can be represented as special cases of the abstract resource accounting model, and therefore can be understood and reasoned about in a uniform manner.

We consider a typical configuration where a *server (cloud) resource* and a *client resource* interact with each other by means of requests/responses (req/res) sent through a communication channel (see Fig. 10).

As shown in the figure, the *client resource* uses the interface of the *server resource* to place its requests and collect

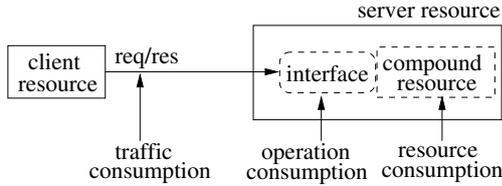


Figure 10. Accounting model of an abstract resource.

the corresponding responses. This deployment incurs three types of consumption charges: **traffic consumption**, **operation consumption** and **resource consumption**. Traffic consumption represents the amount of traffic (for example in MBytes) generated by the requests and responses on the communication channel. Operation consumption captures the activities generated by the client resource on the interface such as the the number of requests (also called operations) and the number of responses produced. Finally, resource consumption represents the actual consumption of the resource measured in units that depend on the specific nature of the resource, for example, in units of volume (for example, MBytes), time or a combination of them (for example, MBytesHours).

As the figure suggests, the accounting model for a given resource is an aggregation of three elementary models: a model for traffic consumption, a model for operation consumption and a model for resource consumption. These elementary models operate independently from each other, thus they can be specified and examined separately. In particular, a provider should make sure that each of the three elementary models are consumer-centric. This should be done by paying attention to the three causes (identified at the beginning of this section) that could lead to discrepancies between the data collected by consumers and providers.

### B. Another Look at S3 and EC2

The accounting models of S3 and EC2 map easily to that of the abstract resource, and permit us to analyse them (from the point of view of consumer-centricity) in a succinct manner. Concerning S3, we can say that the models of the two elementary resources for traffic consumption and operation consumption are strongly consumer-centric, but suffer from incompleteness and ambiguities (that we pointed out earlier) and the model for resource consumption is weakly consumer-centric (checkpointing event is not observable), making the overall model weakly consumer-centric. The accounting model of EC2 is also weakly consumer-centric: the traffic consumption and operation consumption models are strongly consumer-centric (operation consumption model is precisely specified – there is no charge!), but the resource consumption model is weakly consumer-centric because, as we explained with respect to Fig. 7–b, the event that causes a virtual machine instance to progress from pending to running state is not visible to the consumer.

### C. Elastic Block Storage

EBSs are persistent block storage volumes frequently used for building file systems and databases. They support two interfaces: a Web service interface and a block-based input/output interface. The Web service interface can be used by the client to issue (for example, from his desktop application) administration operations, such as *create volume*, *delete volume*, *attach volume*, *detach volume*, etc. The block-based input/output interface can be used by EC2 VMIs and becomes available upon attaching the EBS to the VMI.

Amazon offers EBSs volumes of 1GB to 1 TB. Upon request, EBSs can be allocated to a client and can be attached to VMIs. The storage space of an EBS becomes available when the clients creates the volume and is released when it is explicitly deleted by the client. During this time period, the EBS can be attached and detached several times and to different VMIs but only to one at a time.

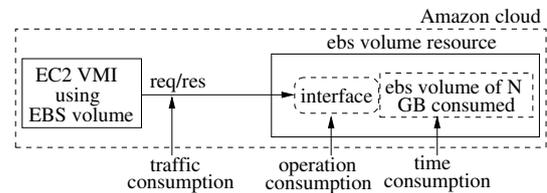


Figure 11. EBS accounting model.

The accounting model for EBS is shown in Fig. 11. Omitted from the figure is the communication channel that the client uses to issue administration operations to the EBS.

In principle and as shown in the figure, an EBS incurs traffic consumption. However, currently Amazon does not charge for this traffic. Operation consumption is measured as the number of input/output operations that the EC2 VMI places against the EBS. Resource consumption is measured in units of time (for example, hrs) and is determined as the time that elapses between the creation and deletion of the EBS. The reason for this is that the amount of storage consumed by the client is determined at EBS creation time.

The EBS accounting model is weakly consumer-centric, because the accounting model for operation consumption includes unobservable events: as Amazon point out in their documentation, the exact number of disk input/output operations cannot be determined accurately by clients because of caching that takes place within applications and operating systems. Fortunately, the number of input/output operations as "seen" by a client is likely to be less than the actual numbers, so the discrepancy always favours the client.

### D. Verifying Billing Charges

If a consumer is using cloud resources then they need to understand how a given deployment will be charged. Ideally, consumers should be in a position to verify the

charges billed to them. In turn this requires taking into consideration the particularities (for example, geographical location of resources) of the deployment and the provider’s current pricing policies (for example, VMI to VMI traffic is free). We believe that the abstract resource accounting model provides a good starting point for developing a tool that can take deployment configuration information and pricing policies to compute billing charges. We suggest this as a direction of future work, and use the hypothetical deployment shown in Fig. 12 for the sake of illustration.

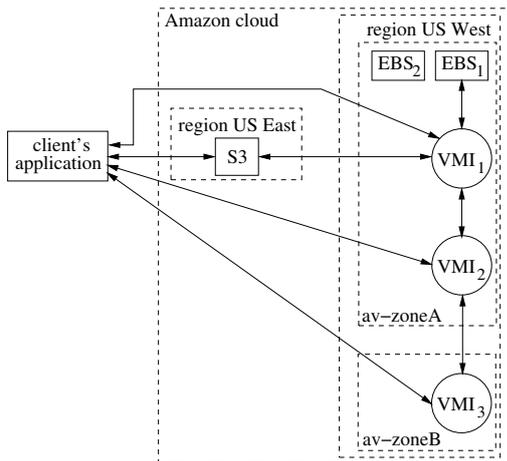


Figure 12. Resource deployment.

The deployment of Fig. 12 involves the client’s application and three types of Amazon basic resources: S3 storage, EC2 VMIs and EBS volumes. It also involves two Amazon regions (US East and US West) and two availability zones (av-zoneA and av-zoneB) located within the US West region. Amazon cloud is divided into regions which are physical locations geographically dispersed (e.g. US-East in Northern Virginia, US-West in Northern California, EU in Ireland). The EC2 cloud is divided in zones which are failure-independent data centers located within Amazon regions and linked by low latency networks.

The arrowed lines represent bi-directional communication channels. Omitted from the figure are the communication channels used by the client to issue administrative commands to the VMIs (*launch, stop, reboot, etc.*) and the EBS (*create volume, attach volume, etc.*).

We open this discussion with a study of the charges that applies to  $EBS_1$  and  $EBS_2$ . Imagine for the sake of argument that they are volumes of 50 GB and 100 GB, respectively. Of concern to us here is the operation consumption and time consumption of the EBSs.  $EBS_1$  will be charged for the number of input/output operations that the  $VMI_1$  places against the  $EBS_1$  interface and also for the period of time of usage of the allocated 50 GB. Being currently detached, the charges for  $EBS_2$  are simpler to calculate, they will consider only the time consumption for 100 GB.

In general, Amazon charges for traffic in (Data Transfer-In) and out (Data transfer-Out) of the Amazon

cloud and for traffic in and out of the EC2 cloud. However, Amazon does not charge for traffic between a VMI and another resource (say S3 storage) located within the same region. Neither do they charge for traffic between two VMIs located within the same availability zone. However, Amazon charges for inter-region traffic between a VMI and another resource (for example, S3) located within a different region. In these situations, the sender of the data will be charged for Data Transfer-Out whereas the receiver will be charged for Data Transfer-In.

With these pricing policies in mind, let us study the charges for  $VMI_1$ . Of concern to us here is traffic consumption and resource consumption.  $VMI_1$  will be charged for inter-region traffic (Data Transfer-In and Data Transfer-Out) consumed on the channel that links it to S3. In addition,  $VMI_1$  will be charged for traffic (Data Transfer-In and Data Transfer-Out) consumed on the channel that links  $VMI_1$  to the client application, as the latter is outside the Amazon cloud. There are no charges for traffic consumed by the interaction against  $EBS_1$  as traffic consumed by the interaction between VMIs and EBSs is free. Neither are there charges for traffic consumed by the interaction against  $VMI_2$  since  $VMI_1$  and  $VMI_2$  share availability zone A. Resource consumption of  $VMI_1$  will be counted as the number of hours that this instance is run.

The charges for  $VMI_2$  will take into account traffic consumption and resource consumption. The traffic consumed will be determined by the amount of Data Transfer-Out and Data Transfer-In sent and received, respectively, along two channels: the channel that leads to the client’s application and the one that leads to  $VMI_3$ . There are no charges for traffic consumed on the channel that leads to  $VMI_1$  because the two instances are within the same availability zone. Again, resource consumption will be counted as the number of instance hours of  $VMI_2$ . The charges for  $VMI_3$  can be calculated similarly to  $VMI_2$ .

We can visualise that S3 will incur charges for traffic consumed on the channel that links it to  $VMI_1$  and on the channel that links it to the client’s application. In addition, S3 charges will account for operation consumption counted as the aggregation of the number of operations placed against S3 by the client’s application and  $VMI_1$ . In addition, the charges will take into consideration resource consumption (storage space consumed) measured in storage-time units. This will be counted as the aggregated impact of the activities (*put, get, delete, etc.*) performed by the client’s applications and  $VMI_1$ .

## VII. CONCLUDING REMARKS

‘Pay only for what you use’ principle underpins the charging models of widely used cloud services that are on offer. An important issue then is the accountability of the resource usage data: who performs the measurement to collect resource usage data—the provider, the consumer, a trusted third party (TTP), or some combination of them? Currently, consumers have no choice but to take whatever usage data made available by the provider as trustworthy.

This situation motivated us to propose the notion of consumer centric resource accounting model. An accounting model is said to be weakly consumer-centric if all the data that the model requires for calculating billing charges can be queried programmatically from the provider. An accounting model is said to be 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); in effect, this means that a consumer (or a TTP) should be in a position to run their own measurement service. We evaluated infrastructure level resource accounting models of a prominent cloud service provider (Amazon) and found that they are only weakly consumer-centric. We presented ideas on how accounting models should be constructed so as to make them strongly consumer centric. We also suggested a systematic way of describing resource accounting models so that they can be understood and reasoned about by consumers.

Service providers can learn from our evaluation study to re-examine their accounting models. In particular, we recommend that a cloud provider should go through the exercise of constructing a third party measurement service, and based on that exercise, perform any amendments to the model, remove potential sources of ambiguities in the description of the model, so that as far as possible, consumers are able to collect with ease their own usage data that matches provider side data with sufficient precision.

#### ACKNOWLEDGMENT

The second author was funded by EPSRC grant KTS-EP/H500332/1.

#### REFERENCES

- [1] A. Mihoob, C. Molina-Jimenez, and S. Shrivastava, "A case for consumer-centric resource accounting models," in *Proc. IEEE 3rd Int'l Conf. on Cloud Computing(Cloud'10)*, 2010, pp. 506–512.
- [2] —, "Consumer side resource accounting in the cloud," in *Proc. 11th IFIP WG 6.11 Conf. on e-Business, e-Services, and e-Society (I3E 2011)*, 2011, pp. 58–72.
- [3] E. Elmroth, F. G. Marquez, D. Henriksson, and D. P. Ferrera, "Accounting and billing for federated cloud infrastructures," in *The Eighth Int'l Conf. on Grid and Cooperative Computing*, Aug. 27–28, Lanzhou, Gansu, China, 2009, pp. 268–275.
- [4] B. Bhushan, M. Tschichholz, E. Leray, and W. Donnelly, "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*, 2001, pp. 107–121.
- [5] E. de Leostar and J. McGibney, "Flexible multi-service telecommunications accounting system," in *Proc. Int'l Network Conf. (INC'00)*, 2000.
- [6] V. Sekar and P. Maniatis, "Verifiable resource accounting for cloud computing services," in *Proc. 3rd ACM workshop on Cloud computing security workshop (CCSW'11)*, 2011, pp. 21–26.
- [7] J. Skene, F. Raimondi, and W. Emmerich, "Service-level agreements for electronic services," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 288–304, March/April 2010.
- [8] C. Molina-Jimenez, N. Cook, and S. Shrivastava, "On the feasibility of bilaterally agreed accounting of resource consumption," in *1st Int'l Workshop on Enabling Service Business Ecosystems (ESBE08)*, Sydney, Australia, 2008, pp. 170–283.
- [9] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou, "Distributed systems meet economics: Pricing in the cloud," in *Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10)*, 2010.
- [10] R. V. den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads," in *Proc. IEEE 3rd Int'l Conf. on Cloud Computing(Cloud'10)*, 2010, pp. 228–235.
- [11] B. Suleiman, S. Sakr, R. Jeffery, and A. Liu, "On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure," *J Internet Serv Appl*, DOI 10.1007/s13174-011-0050-y, Dec 2011.
- [12] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "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.
- [13] M. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon s3 for science grids: a viable solution?" in *Int'l Workshop on Data-Aware Distributed Computing (DADC'08)*, Jun 24, Boston, USA, 2008, pp. 55–64.
- [14] Amazon, "Amazon simple storage service. developer guide, API version 2006–03–01," 2006. [Online]. Available: [www.amazon.com](http://www.amazon.com)
- [15] —, "Amazon elastic compute cloud (amazon ec2)," 2011. [Online]. Available: [aws.amazon.com/EC2/](http://aws.amazon.com/EC2/)
- [16] —, "Amazon ec2 pricing," 2011. [Online]. Available: [aws.amazon.com/ec2/pricing](http://aws.amazon.com/ec2/pricing)
- [17] —, "Amazon ec2 faqs," 2011. [Online]. Available: [aws.amazon.com/ec2/faqs](http://aws.amazon.com/ec2/faqs)
- [18] —, "Amazon elastic compute cloud user guide (api version 2011–02–28)," 2011. [Online]. Available: [docs.amazonwebservices.com/AWSEC2/latest/UserGuide/](http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/)
- [19] J. Murty, *Programming Amazon Web Services*. O'Reilly, 2008.