Challenges with Tenant-Specific Cost Determination in Multi-Tenant Applications

Anna Schwanengel and Uwe Hohenstein
CT RTC ITP SYI-DE, Siemens AG
Munich, Germany
{anna.schwanengel.ext, uwe.hohenstein}@siemens.com

Abstract— **One key element to make Software-as-a-Service (SaaS) successful is so called multi-tenancy, which refers to an architecture model where one software instance serves a set of multiple clients of different organizations (tenants). Hence, it reduces the number of application instances and, in that way, operational costs in a Cloud. The problem SaaS providers are faced within everyday's business is how to define a billing model that has the chance to make profit in a public Cloud. Being profitable with SaaS, the art is to bill tenants in such a way that covers the costs for resources for the underlying PaaS/IaaS provider. This paper discusses some challenges with metering the consumption of tenants as a prerequisite for defining a profitable billing model.**

Keywords - Software-as-a-Service; Multi-Tenancy; Billing; Costs; Resource Utilization

I. INTRODUCTION

Since many years a paradigm shift how software is delivered to customers occurs. It changed from installing developed software applications at the customer in-house and operating it on-premise, to a more consumer-based model. Software became an on-demand service drawn from the Internet, i.e., Software-as-a-Service (SaaS) [1]. SaaS is a delivery model that enables customers to rent services without local installation and license costs.

In this context, multi-tenancy is a key element to achieve a successful SaaS business, though not being the guarantor for more revenue. Multi-tenancy means multiple tenants from different organizations share a system operated by one company. The respective application is used by several tenants of a SaaS provider [2]. Thereby, each tenant serves plenty of users who actually use the software. A multi-tenant architecture postulates that the application is able to partition its data and procedures virtually. Each tenant gets a virtual instance, which can be customized according to his wishes, running on the same physical instance, while not being influenced or even aware of the other tenants working concurrently.

In single-tenant systems, each tenant obtains its own instance running the application (or database), which reduces management efforts regarding the mapping of the resources to each tenant. However, looking at the overall efficiency, one can observe some drawbacks, as in a lot of cases many server instances will be low utilized at most time points [3]. This system utilization can be improved by operating a multi-tenant service, where fewer instances are used to serve tenants in a shared environment. Moreover, operational costs can be saved when the SaaS provider deploys an application on the PaaS or IaaS layer of a Cloud provider. A SaaS provider pays for the resources his SaaS application uses. That means being charged by CPU time, number of transactions, database space etc. The more payable resources are shared, the less costs an application produces. One important aspect is to design the architecture in a way that uses the resources efficiently [4].

In this paper, we focus on another economical problem of SaaS providers, which has been paid less attention in the research area. On the one hand, we have *cost models* defined by IaaS/PaaS providers, a SaaS provider has to pay for when running applications. But a SaaS provider has also to define a *billing model* to charge his tenants for application usage. Both models have to be balanced in a way that SaaS providers obtain a suitable return of investment and are able to make profit while having an attractive billing model for tenants. The investment covers both, the Cloud operational costs and costs for application development or SaaS-enabling of existing applications.

We are approaching this aspect from a technical view. A lot of billing methods have been discussed in the literature such as pay-as-you-go, pay-per-user, pay-per-feature, or a fixed monthly fee [5]. All have in common that a SaaS provider has to keep an overview over total costs *and* tenant-specific costs in order to offer a profitable billing model. Section II stresses this point and motivates the need for tenant-specific metering of resource consumption.

We present challenges for SaaS providers to balance outgoing costs for the underlying PaaS/IaaS provider and ingoing revenue from the tenants. We choose Windows Azure for this investigation because of its PaaS offering that ships with a complete development and deployment environment. There are no problems with product licensing, as this is part of the platform and the cost model, which makes the cost calculation easier – see Section III.

Section IV gives some insight into cost reasoning for multi-tenancy within Azure. Section V discusses what technical concepts of Azure can be used to monitor tenant-specific resource consumption. A prerequisite, how tenants can be identified, is explained in Section VI. Section VII provides an overview of related work in the multi-tenancy area before Section VIII concludes and names future work.

II. PROBLEM SPACE

It is commonly agreed that a well-economical SaaS provider has to support multi-tenancy, i.e., giving tenants a tailored, best-fitting application satisfying their specific

 ISBN: 978-1-61208-271-4

requirements by customization, while sharing as much resources as possible to achieve higher capacity utilization. Thereby, SaaS provider have to reflect upon easy implementation (as in single-tenant systems, where every tenant holds its own application) and costs (which is more adjusted in multi-tenant systems serving all tenants by one instance). That is in accord with economy of scale sharing both the underpinning infrastructure as well as the hereon running software. This point can also be seen in [6] and [7], where several architectures are distinguished regarding what is shared by tenants: the topmost web frontend, middle tier application servers, and underlying database. Nevertheless, when supporting all tenants by one instance in a multi-tenant system, the question is how to charge each tenant, while targeting at profit. Defining a billing model is easy but how to monitor whether it is reasonable?

Several billing models have been proposed. Most of them are post-paid models. Thereby the tenant receives a bill and pays for usage periodically [8]. To invoice the consumption costs, usage of each tenant is observed and aggregated [9]. The safest method from a SaaS provider's perspective is to charge tenants the same pay-as-you-go way as PaaS/IaaS providers do for their resources, i.e., OPEX are directly forwarded to tenants, plus an additional charge. Such a model is very technical and not cost-transparent for tenants. From a SaaS provider's view, this situation is complicated when several tenants are served by one instance. Therefore, it is important to estimate or even compute the resource costs (e.g., for consumed storage, or CPU), in particular how many resources one tenant uses. This implies the monitoring of each tenant and logging the way they use the application. More precisely, it requires observing the resource usage of the applications for each tenant, and raising an invoice based on usage metrics.

Alternatively, billing models can be based upon factors that are better understandable by tenants, like usage time. The problems for SaaS providers remain the same, and the Cloud cost model must be transformed to a billing model.

A SaaS provider can also charge its tenants by a fixed rate, e.g., per month. However, it is difficult to predict the costs a tenant's usage will produce. Moreover, exhaustive usage by one tenant could reduce the SaaS providers' revenue, even to minus. On these grounds, a precise cost control of each tenant can be used to throttle frequent users to reduce this risk – if SLAs are defined accordingly.

In a pay-per-user billing model, users must be registered and the number is then known. However, there is again a risk of undercharging over-utilizing tenants.

Billing may also be conducted in a pre-paid method. Pre-paid clients load a deposit onto their accounts previous to any consumption. During the usage, this credit is debited and in case of reaching a limit, the tenant has to reload money for service use. Although the pre-paid model sounds promising to SaaS providers offering profit-ability, the post-paid model is more common. Anyway, one has to check whether a tenant's limit has been reached.

All this comes along with a big problem for the SaaS provider: he has no clue whether his offering is profitable. A detailed monitoring of costs produced by tenants is necessary, independent of the billing method. Besides, cost models of IaaS/PaaS providers are quite complex and take technical parameters into account. This makes it not only difficult to estimate the costs for a given application [10], but also to derive costs for each tenant. The different cost factors that PaaS/IaaS providers charge (which differ enormously from provider to provider) make it difficult to run a clear-cut course. Several systems (e.g., EC2) bill according to a usage-of-instance charge and raise the price additionally based on the absolute number of transferred bytes and not adapted on duration or network activity [11].

Another aspect, which requires closer attention, is that an overview of the total amount of used resources and resulting costs is usually only given on a monthly basis. With only getting a monthly bill from a PaaS/IaaS provider with an aggregated cost report over the consumed resource capacity for his tenants, a SaaS provider could not get any detailed data about the cash accounting. Thus, a SaaS provider could not counteract in time, when his service is getting unprofitable by tenants with frequently active users. There is a strong need for a tenant-specific accurate cost model, which is required for:

- a consumption-based model that charges back tenants for their consumed resources;
- a tenant-specific profit-making check, which illustrates, whether the chosen business model for one/all customer(s) is appropriate to make profit;
- a timely reaction in order to throttle frequent and too expensive tenants; Throttling just at the end of a month will be too late to compensate losses.

This paper deals with these challenges of estimating costs on a per-tenant basis. In particular, costs have to be conducted in an efficient manner that only means a minimal amount of extra burden, to avoid latency and costs.

III. MICROSOFT AZURE AND ITS COST FACTORS

Since we base our investigation on a concrete PaaS platform, Microsoft Azure, we here briefly present basic concepts and the cost model of the Azure Cloud platform according to the status quo when writing this paper [12].

Compute instances (VMs including equipment), called Web and Worker Roles, are charged for the number of hours they are deployed. As seen in Table I, there are several instance categories: A *small* instance (default) costs $0.12 per hour; the more powerful *medium*, *large*, and *extra large* instances have twice the price as the preceding category, i.e., an extra large instance is charged for $0.96 per hour (i.e., factor 8 compared to a small instance). The instance categories scale in a linear manner with regard to equipment. That is, a medium instance (M)

TABLE I: PRICES FOR COMPUTE INSTANCES

	CPU	RAM	HDD (GB)	MBps	$ / h	I/O performance
XS	Shared	768MB	20	5	0.04	Low
S	1,6GHz	1,7 GB	225	100	0.12	Moderate
M	2 x	3,5 GB	490	200	0.24	High
L	4 x	7 GB	1000	400	0.48	High
XL	8 x	14 GB	2040	800	0.96	High

 ISBN: 978-1-61208-271-4

has double of CPU, disk etc. than a small instance (S) resulting in a double price. The exception is an *extra small* instance (XS) category. The prices are taken on an hourly basis. Even if a compute instance is used for only 5 seconds, a full hour has to be paid.

For Azure table, blob and queue storages, costs depend on bandwidth, storage consumption and transactions.

Storage is billed based upon the average usage during a billing period. If, e.g., 10 GB of storage are used for the first half of a month and none for the second half, 5 GB of storage are billed for average usage. Azure measures the consumption at least once a day. Each GB of storage is charged with $0.07. Please note that storage consumption takes into account the physical storage, which consists not only of raw data; the length of property names, and the property data types also affect the size of actual data [13].

Any access to storage by transactions has to be paid: 100,000 transactions cost $0.01. Bulk operations, which bundle inserts, count as one transaction.

The outbound transfer to the North America and Europe regions is charged with $0.12 per outgoing GB, the Asia Pacific Region is more expensive. It is important to note that the transferred data has some typical XML overhead according to the protocol. Data transfer is for free within the same affinity group, e.g., for compute instances that run in the same data center. All inbound data transfers to the Azure Cloud are also at no charge.

The costs for an Azure SQL Database, a virtualized SQL Server, are also based on monthly consumption. Up to 100 MBs are charged with $4.995 a month. Up to 1 GB, the overall price is $9.99. Any GB exceeding 1 GB costs $3.996. Having reached 10 GB, the prices again decrease to $1.996 per additional GB, and beyond 50 GB, a GB costs only $0.999. This means, a 10 GB is charged with $45.954: $9.99 for the first GB, and 9 * $3.996 for the remaining 9 GB. Azure instance is charged monthly for the number of databases and amount of data used a day. Further charged services exist, e.g., for authentication by Azure Access Control, but they are out of scope here.

These cost factors are important for SaaS providers to determine the price for a deployed application in a rented PaaS/IaaS environment. Knowing the precise costs for the SaaS application is the core element when a SaaS provider forms a billing model for its tenants. Only in this case, the SaaS provider can create an economical billing method of accounting with high profit.

IV. REASONING FOR MULTI-TENANCY

Multi-tenancy is often presented as a solution to make profit or to deploy SaaS applications economically. The statement is more or less generally accepted. Anyway, we want to provide some calculations to show the effect of multi-tenancy in case of Microsoft Azure.

At first, we consider storing data in an Azure SQL Database. The costs are primarily based on storage consumption. But there is no cost difference between storage in one or in several databases, no matter whether placed on one database server. Hence, there seems to be no cost-benefit for sharing one database or server between several tenants. Hence, a question is arising: Are several databases (one per tenant) really more expensive than keeping all tenants' data in one large database?

First, pricing in Azure occurs in increments of 1 GB. Thus, four 1.1 GB databases are charged with 4*2 = 8 GB, i.e., 8 * $9.99 = $79.92 a month, while a single database of 4.4 GB is charged with 5 GB. Next, the storage price decreases with the size. Assume there are 4 tenants with databases à 3.1 GB, 4.3 GB, 38.3 GB, and 87.2 GB respectively. The monthly storage costs for having for each tenant a database of its own are:

3.1 GB: 1*$9.99 (1st GB) + 3*3.996 = $ 21.978
4.3 GB: 1*$9.99 (1st GB) + 4*3.996 = $ 25.974
38.3 GB: $45.954 (1st 10 GB) + 19*$1.996 = $ 83.878
87.2 GB: $125.874 (1st 50 GB) + 38*$0.999 = $163.836

This is in total $295.666. In contrast, a single database for all the 132.9 GB costs

$125.874 (1st 50 GB) + 83*$0.999 = $208.791.

This means a 26% cost reduction of $87. However, that rough comparison does not take into account that record sizes increase slightly for the one-in-all database due to the *tenantID* for distinguishing tenants. Keep also in mind that there is a limitation of 150 GB per database, which hinders putting a higher amount of tenants with larger storage consumption in one database!

The constellation is similar for table storages, albeit, the cost decrease is much lower: Here, 1 GB costs 7ct. Any additional GB exceeding 1 TB is charged with 6.5ct. Beyond 50 TB, the price is 6ct. Storing 10 TB in ten 1-TB tables ($700) makes a difference to one 10-TB table ($655). This plays a role only for larger data volumes.

For compute instances, 12 ct per hour are charged for a small instance, i.e., $1,051.20 per year. Saving instances by sharing services is, therefore, reasonable. There is a real cost difference when the provider could serve ten tenants with one instance ($1,051.20) instead of giving each tenant an instance of its own (10 * $1,051.20).

We are faced with an additional hard decision in determining whether to rent a higher amount of less capable computing instances or to take rather fewer high performing instances. From the cost's view point at a first glance, it makes no difference whether a SaaS provider rents four small (S) instances or one large (L) instance; the SaaS provider has to pay the same price for the same capacity. However, if additional instances are required, due to heavy load by tenants or serving an increasing number of new tenants, a SaaS provider has to add extra instances. In this process, however, the type of instance (XS, S etc.) is already determined at deployment of the application. If applications are designed for L instances, the SaaS provider has to start a further L instance, even though a cheaper S instance would have been sufficient to serve the additional tenants' users. That will result in a less profitable service provisioning and in less revenue. Generally, constant system utilization is improbable and high variations in service usage often occur [14].

 ISBN: 978-1-61208-271-4

V. CHALLENGES FOR TENANT-SPECIFIC COST ESTIMATION

In Section II, we motivated why monitoring tenant-specific consumption costs are useful. In this section, we discuss the features Microsoft Azure provides to this end, thereby concentrating on real multi-tenant systems with tenants sharing instances. We give some insight in what support is available, to what extent, and what is missing.

A SaaS provider has only some basic support by Azure. He gets a bill once a month for the monthly consumption of all cost factors: CPU time, storage, database units, outgoing data transfer, and number of transactions. Furthermore, there is a management API giving access to the recent deployment including information about the number of instances of what size and the starting time. Enabling performance counters allows for tracking aggregated usage for Blobs, Tables and Queues [15]. Please note all this is consolidated for one storage account; the limit is 5 for Azure subscribers. Hence giving each tenant a subscription of his own is unfeasible.

The structure of this section follows the Azure cost factors and distinguishes some multi-tenancy approaches.

A. Azure SQL Database

1) Each tenant obtains a physical DB of its own

In this case, the database size can easily be determined by means of a SQL query using the dictionary information. However, one important question remains: When should be the consumed storage measured?

The cost model says that the storage consumption is measured once a day by Azure, but the time point is unknown, because Azure argues that the charge amortizes during the month. However, the storage consumption might vary a lot day by day and in fact within one day. According to this, even if we periodically check the consumption each day, we do not know when Azure is measuring, and this is relevant for our bill. If we take the values for the consumption at noon, the consumption might be completely different to midnight; maybe this is the time Azure measures our occupancy. To solve this point, Azure's internal measuring must be laid open.

In addition to the storage consumption, Azure also charges for the outgoing data transfer. Outgoing means leaving the data center. This cost fact can be ignored unless the SaaS application offers tenants a direct access to the database, which is albeit rather unusual, e.g., due to isolation and security issues [16].

2) Tenants share a common database

If a common database is shared by multiple tenants, it is more difficult to determine a tenant's part of the database. Assuming that each tenant is maintained by a unique tenant identifier (*tenantID*), it is possible to count the number of records in each table in order to get a rough impression. Nevertheless, this number does not reflect the storage consumption since the length of records might vary from tenant to tenant. A more complex and time-consuming query can sum up the length of all values. Furthermore, the storage for indexes remains unknown.

Moreover, the same questions as above remain about when to measure the numbers for database consumption; we again do not have any information at which time point Azure's measurement takes place.

B. Azure Table Storage

The table storage usage is charged by outgoing data transfer, memory usage, and the number of transactions.

1) Each tenant obtains a physical table set of its own

Unfortunately, there is no efficient way to measure the physical table size. The management API does not yield concrete measurements or consumption numbers, but only a monthly summary for a complete storage account (with several tables). To counteract this lack in tenant-specific billing, some solutions are possible, even though problems remain. First, tenant records can be counted, which means accessing the complete table. This can raise transactional costs, and performance impacts may occur. Besides, still some uncertainly remains due to unknown record sizes.

A more efficient approach is to enumerate records during insert. Then, we are able to ask for the latest record by a timestamp-query; this is approximately the number of records. However, we have no numbers for already deleted records. More cost-intensive is to maintain two counters for insert and delete operations, which doubles the transactional costs. Nonetheless, the number of records is only a rough estimation, and the problem how to compute the specific record sizes still remains.

Consequently, there is a strong need to add further tracing for tenant-specific storage actions. A modular possibility may be to use aspect-orientation to intercept operations [17], however, we are then only able to measure accesses via the C# storage library, but cannot quantify REST calls to the storage. A simpler form is to register event handlers for inserts, which is a rather rudimentary, limited mechanism. When implementing event handlers to observe storing and deleting operations in the table storage, the event handler requires the *tenantID* as a prerequisite for enabling a tenant-specific billing. Anyway, the best way is to add some kind of monitoring in the application, whereby one important problem still remains: When to measure the tenant's consumption?

2) Tenants share a common table storage

In the case of tenants sharing a table, we find the same problem as above. We have to query complete tables to count records, now for one tenant. The counting can be conducted more efficiently if the *tenantID* is taken as the *PartitionKey*. Then, calculation can be done in one partition, reducing search space and raising performance.

3) Transactions

Another cost factor is the number of transactions on the table storage. Charging 100,000 transactions with 1ct appear like micro-costs at a first glance. But investigations show that transactions could be the dominating cost factor in Azure [10]. Moreover, the term transaction must be taken carefully. Every operation to the storage, even asking for the list of tables, is considered as a transaction. Some operations can be performed in bulks; each bulk is

 ISBN: 978-1-61208-271-4

then a transaction. And finally, each query is a transaction whereby a continuation token is returned if the result is too large or runs too long. Then, successive queries become necessary, which are counted as transactions as well.

There exist performance counters (Azure storage metrics) which however track only the number of transactions for one storage account. This might be an efficient way to compute the overall transactions on a daily basis, but does not yield any tenant-specific information. Further, tracking the number of transactions must again be done by introducing specific tracing in order to get precise data.

4) Outgoing data transfer

The final cost factor is the outgoing data transfer (leaving the data center). These costs are presumably irrelevant unless queries on the table storage are directly performed by tenants, which is rather unusual.

C. Blob and Queue Storage

Principles and techniques for handling cost aspects for blobs and queues are quite similar to Azure table storage and the same mechanisms as explained in *B* can be applied. However, the queue storage consumption seems to be irrelevant since queues will usually not keep large amounts of data, unless there is some congestion in the system. The dominant cost factor will be the transactions.

D. Compute Instances

In Azure, computing power is organized by means of Web and Worker Roles, as described in Section III. The major cost factor is the number of hours a role runs. Any application can be distributed over several Web and Worker Roles. Furthermore, an application can scale out by setting up additional instances of an implemented role to handle sporadic load peaks [14] with a load balancer.

The Azure management API yields some information that can be used to monitor costs such as the size of a role (S, M, L etc.), the number of instances for each role, their status (running, suspended etc.), the starting time, etc. In principal, it is enough to poll the data when the current consumption is needed. However, we are not aware of removed instances and roles since they silently disappear from the report. In order to get notice of any decrease of instances, it is necessary to poll periodically. Some uncertainty remains as an instance can run only for one minute, being charged with one hour. This event will presumably get lost unless we check within that minute.

Generally, this data does not reveal any tenant-specific information; it just shows values of the overall consumption of a multi-tenant application. If there is a relationship between Web/Worker Roles and tenants, such a separation would be possible, however, thwarting principles of multi-tenancy. To obtain tenant-specific information, additional logging should monitor the number of requests. This kind of data is available in performance counters, but again only covers the whole application.

Please note, there is no obvious relationship between VM operation costs and how much a tenant contributes to these by measuring CPU time etc. Hence, these are only rough indicators for a tenant's portion of usage.

E. Further Notes

It is important to note that measurements themselves could affect the costs. Consequently, there is a trade-off between collecting precise data and being cost-efficient. This basically concerns the frequency of periodical measurements, the efficiency of queries etc.

VI. DETERMINING A TENANT

One important issue for the previous discussion is how to extract a tenant, which uses the application, from the service URL. The following discussion summarizes relevant aspects. Thereby, we investigate four ways of defining SaaS URLs [18] and how to extract a tenant.

A. Using a General URL

A SaaS provider may offer a general URL in the manner of *http://www.SaaSprovider.com*. Each tenant has to register all of his users for the specific services with user and password; particularly, each user obtains a unique tenant identifier (*tenantID).* The assumption is that each user is exclusively associated with a single tenant. Using a service such as *http://www.SaaSprovider.com/Service1*, a tenant's user has to log in with his credentials. A central component is then able to determine the user's *tenantID*. While this implementation is rather simple, several fundamental problems are obvious in this approach.

At first, the service itself must be generic and unbranded until the user has logged in. Similarly, a tenant-specific customization can only take place after login. Before login, the service can only be general due to the unknown *tenantID*. As a direct consequence, it is difficult to have more than one identity provider (such as an own Active Directory). The identity provider cannot be known before the tenant is known. But in most cases, tenants want to specify the identity provider fitting to their infrastructure. Next, it is immediately visible that the user is accessing a multi-tenant application because the URL does not contain the tenant. Furthermore, there is no way to allow for anonymous users that have no account and consequently no relationship with a tenant. Hence, SaaS providers are restricted to supporting all solvent users. Finally, a user cannot have a relationship with more than one tenant unless they have different credentials.

To sum it up, although a tenant can easily be identified by picking up the login credentials of the users, this approach has some drawbacks and is unsatisfying. For that reason, we consider further possibilities as following.

B. Tenant Parameter in the URL

As an alternative to the first approach for URL design, the URL can per default contain the tenant's name as an identifier in two different ways, for instance:

- *http:// www.SaaSprovider.com /tenant1*
- *http:// www.SaaSprovider.com?t=tenant1*

Now, the application knows immediately who the accessing tenant is, and customization can take place for a

tenant just as various identity providers are possible for authentication. Furthermore, we can observe the advantage that anonymous users are possible as they do not depend on an identifiable relationship to a tenant. Additionally, users can have accounts with more than one tenant, because their access is scoped by tenant.

Unfortunately, there are still some problems. At first, it is still obvious that this approach is a multi-tenant application, because the URL specifies a host *SaaSprovider* that has no meaning to the user. That is why the user cannot deduce the service, which he actually wants to use, from the given URL. Next, both of the above provided URLs are difficult to guess, i.e., users will be unable to find the application by means of 'URL surfing'. If the user just haphazardly tries out random URLs such as *www.tenant1.com/service1* or *http://service1.tenant1.com*, he will never score a hit, because the service is URL-invisible. It is to note that the URL is an important part of a company's brand. Having URLs such as *http://www. SaaSprovider.com/tenant1* with someone else's host name in a URL (here *SaaSprovider.com*) is only a "second class" branding and insufficient for big companies.

However, identifying a tenant with an ID in the URL is possible by extracting the tenant's name by means of ASP.NET MVC URL Routing.

C. Tenant in a Sub-Domain

A better approach is to embed the tenant identifier (*tenantID*) in the URL as a sub-domain: *http://tenant1. SaaSprovider.com.* Moreover, it is possible to apply a DNS alias to redirect the URL to *www.SaaSprovider.com.* Advantages of this proposal are obvious: It is still possible to identify every single tenant, whilst the URL is branded since the tenant name, *tenant1*, appears directly within the URL, and it is now less obvious that *tenant1* is one of many tenants that are using the application. The URL can be found out with URL guessing and by trial and error.

Even the technical challenge of extracting tenants from the URLs can be solved, since the tenant is passed with the HTTP request in the Host Header, albeit it is more complicated.

D. Tenant in a Domain

Finally, a tenant may use its domain, e.g., *http://www. tenant1.com.* The URL can be mapped to *www.SaaS provider.com* in the tenant's DNS configuration. Here, the tenant can be identified by using the *Request.Url* C# class.

In summary, it can be stated that tenant identification is possible for all four approaches; this is the basis for our considerations to realize tenant-specific billing. However, the approaches are characterized by different quality and accordingly efforts and costs. This has to be considered when deciding how to conduct tenant identification.

VII. Related Work

A lot of research is done in the field of multi-tenancy, where also traditional aspects of distributed computing remain important. Fehling et al. come up with prospects for the optimization of multi-tenants by the distribution of the tenants regarding Quality of Service [18]. Additionally, security and privacy issues should also be regarded for multi-tenancy. To this end, Jensen et al. present an overview of technical security problems [20]. Besides, requirements for efficient multi-tenancy regarding performance or isolation are explored by Guo et al. They present a design and implementation framework to support multi-tenant services [2]. Since multi-tenancy is linked to large client amounts, economic concerns raise importance, too, as providers need to operate with high profit to remain competitive. To reduce overall resource consumptions in multi-tenant environments, [21] introduces a method for implementing cost-efficient multi-tenancy by optimized tenant placement. Also [22] puts values of utilization and performance models in genetic algorithms to reduce there-by costs, albeit, they do not concern tenant-specific billing.

Other researchers consider solutions to implement cost-efficient multi-tenancy, looking at the infrastructure, middleware and application tier, which all can be shared among tenants [23][24]. However, for fault-tolerance, one still needs an existence of the same application on different instances – regardless of the particular tier. So, if an application transparently moves to another instance, this must be traced and considered in the bill to fit a tenant-specific pricing. This problem is not considered there.

In general, providers bill their tenants in different models. The most common pricing models are either the tenants paying a fixed monthly fee, or in a pay-per-use model, where the tenant only pays for the resources he had used, or even the resources may be charged usage-based [25]. With multi-tenancy, SaaS providers' profit may be increased, but on the other hand, one has to monitor each tenant resource usage and relate this to his monthly bill. Therefore, Cheng et al. set up a monitoring framework to trace tenants' allocations at runtime and to observe the performance of each tenant based on the individual SLAs [26]. However, they do not provide a tenant billing model.

Bezemer and Zaidman, discuss, based on existing single-tenant applications, another aspect of costs associated with multi-tenant applications: maintenance efforts. The recurrence of maintenance tasks (e.g., patches or updates) raise operating costs and show the demand of exact planning of maintenance costs, which must be apportioned among the tenants [27][28].

Nevertheless, the profitable aspects for the SaaS providers are researched insufficiently in the field of multi-tenancy. Reflections about their balancing act between making revenue through tenants' charges and paying for the tenants' used capacity at the PaaS/IaaS provider are extremely understudied until now. Therefore, we came up with an overview of the remaining challenges for the SaaS providers, which want to offer their services to multiple tenants in an economical business model.

VIII. Conclusion

In order to save costs and run economical businesses, SaaS providers rely on multi-tenancy, albeit it is no recipe for more revenue. By building multi-tenant applications, a SaaS provider can support multiple tenants from different

 ISBN: 978-1-61208-271-4

organizations with shared instances, being simultaneously used. This and a better utilization by tenants through re-use may lead to higher revenue for SaaS providers.

Within the paper, we depict considerations that enable SaaS providers to succeed in balancing outgoing costs for the PaaS/IaaS resources and ingoing revenue from tenants to operate economical business. We motivate why it is necessary to monitor the detailed costs per each tenant in a more fine-granular manner. We focused on Microsoft Azure and came up with reasoning for multi-tenancy and discussed features of the Azure infrastructure. Until now, SaaS providers receive monthly bills from Azure about the past resource usage by its tenants. This is insufficient because no precise and in time tracking of tenant-specific costs is available. Although some tenant-specific costs can be determined with more or less effort, they might be expensive and lead to additional costs for the SaaS provider. Anyway, for multi-tenant SaaS providers some uncertainty about costs remains and their challenge is still to observe how much a tenant uses of a specific resource type in order to achieve high profitability.

As future work, we plan to also analyze other Cloud platforms such as Amazon IaaS/PaaS regarding its support to trace costs by each tenant. Further, we want to conduct experiments and analyze the corresponding data to give some concrete suggestions how to integrate tenant-specific billing in new and even already existing applications. We will also investigate and compare multi-tenant application built upon a PaaS Cloud and an IaaS platform in order to give an even more precise insight in cost factors. We think the PaaS version will produce more expensive bills, but will also decrease development costs than the IaaS version. Moreover, we work on adequate possibilities for application-specific logging. All this work should finally lead to a consumption-monitoring system.

REFERENCES

[1] A. Dubey and D. Wagle, “Delivering software as a service,” In: The McKinsey Quarterly, 2007, pp. 1-12.

[2] C. Guo, et al., “A framework for native multitenancy application development and management,” Proc. on Enterprise Computing, E-Commerce and E-Services, 2007, pp. 551-558.

[3] B. Wilder, “Cloud Architecture Patterns: Using Microsoft Azure,” Sebastopol: O'Reilly Media Inc., 2012, pp. 77-79.

[4] R. Buyya, C. S. Yeo, and S. Venugopal, “Market-oriented clouds: Vision, hype, and reality for delivering IT services as computing utilities,” Proc. on HPC, 2008, pp 5-13.

[5] http://apprenda.com/library/software-on-demand/saas-billing-pricing-models [retrieved: March 2013]

[6] S. Walraven, E. Truyen, and W. Joosen, “A middleware layer for flexible and cost-efficient multi-tenant applications,” Proc. 12th Middleware, 2011, pp. 370-389.

[7] S. Walraven, E. Truyen, and W. Joosen, “Towards performance isolation in multi-tenant SaaS apps,” Proc. on Middleware for Next Generation Internet, 2012, pp.1-6.

[8] M. Lindner, F Galán, and Clovis Chapman, “The cloud supply chain: A framework for information, monitoring, accounting and billing,” Proc. on Cloud Computing, 2010.

[9] I. Ruiz-Agundez, Y.K. Penya, and P. Bringas, "A flexible accounting model for clouds," SRII, 2011, pp. 277 - 284.

[10] U. Hohenstein, R. Krummenacher, L. Mittermeier, and S. Dippl, “Choosing the right cloud architecture - A cost perspective,” Proc. on Cloud Computing and Services Science (CLOSER), 2012, pp.334-344.

[11] S. Seetharaman, “Energy conservation in multi-tenant networks through power virtualization,” Proc. on Power aware computing and systems, USENIX, 2010, pp. 1-8.

[12] Azure Pricing, 2013, www.windowsazure.com/en-us/pricing/details [retrieved: March 2013]

[13] B. Calder, “Windows Azure Storage billing,” http://blogs.msdn.com/b/windowsazurestorage/archive/2010/07/09/understanding-windows-azure-storage-billing-bandwidth-transactions-and-capacity.aspx [retrieved: March 2013]

[14] A. Schwanengel, U. Hohenstein, and M. Jäger, “Automated load adaptation for cloud environments in regard of cost models,” Proc. on CLOSER, 2012, pp.562-567.

[15] J. Haridas, M. Atkinson, and B. Calder, “Azure Storage metrics,” http://blogs.msdn.com/b/windowsazurestorage/archive/2011/08/03/windows-azure-storage-metrics-using-metrics-to-track-storage-usage.aspx [retrieved: Mar. 2013]

[16] J. Schroeter, S. Cech, S. Götz, C. Wilke, and U. Aßmann, “Towards modeling a variable architecture for multi-tenant SaaS applications,” Proc. on Variability Modelling of Software-Intensive Systems, 2012, pp. 111-120.

[17] U. Hohenstein and M. Jaeger, “Using Aspect Orientation in Industrial Projects: Appreciated or Damned?”, Proc. on Aspect-Oriented Software Development, 2009, pp.213-222.

[18] Designing Multitenant Applications on Windows Azure: http://msdn.microsoft.com/en-us/library/windowsazure/hh689716.aspx [retrieved: March 2013]

[19] C. Fehling, F. Leymann, and R. Mietzner, “A Framework for Optimized Distribution of Tenants in Cloud Applications,” Proc. on Cloud Computing, 2010, pp. 252-259.

[20] M. Jensen, J. Schwenk, N. Gruschka, and L. Iacono, „On technical security issues in cloud computing,” Proc. on Cloud Computing, 2009 pp. 109-116.

[21] T. Kwok and A. Mohindra, “Resource calculations with constraints, and placement of tenants and instances for multi-tenant SaaS applications,” Proc. on Service-Oriented Computing, 2008, pp. 633-648.

[22] D. Westermann and C. Momm, "Using software performance curves for dependable and cost-efficient service hosting," Proc. on Quality of Service-Oriented Software Systems (QUASOSS), 2010, pp. 1-6.

[23] C. Osipov, G. Goldszmidt, M. Taylor, and I. Poddar, “Develop and deploy multi-tenant web-delivered solutions using IBM middleware: Part 2: Approaches for enabling multi-tenancy,” In: IBM’s technical Library, 2009.

[24] C. Momm and R. Krebs, “A qualitative discussion of different approaches for implementing multi-tenant SaaS offerings,” Proc. Software Engineering 2011, pp. 139-150.

[25] M. Armbrust, et al., “A view of cloud computing,” Communications of the ACM, 53(4), April 2010, pp. 50-58.

[26] X. Cheng, Y. Shi, and Q. Li, “A multi-tenant oriented performance monitoring, detecting and scheduling architecture based on SLA,” Proc. on Joint Conferences on Pervasive Computing (JCPC) 2009, pp. 599-604.

[27] C. Bezemer and A. Zaidman, “Multi-tenant SaaS apps: Maintance dream or nightmare,” In: Technical Report of Delft Uni. of Technology, TUD-SERG-2010-031, 2010.

[28] C. Bezemer, A. Zaidman, B. Platzbeecke, T. Hurkmans, and A. Hart, “Enabling multitenancy: An industrial experience report,” In: Technical Report of Delft Uni. of Technology, TUD-SERG-2010-030, 2010.

 ISBN: 978-1-61208-271-4