Aquarium: An Extensible Billing Platform for Cloud Infrastructures

Georgios Gousios Christos KK Loverdos Panos Louridas Nectarios Koziris
Greek Research and Technology Network (GRNET)

Abstract

An important part of public IaaS offerings is resource management and customer billing. In this paper we present the design and implementation of Aquarium, an extensible billing service software. Aquarium associates state changes in cloud resources with respective charges, based on configurable, user-specific and versioned charging policies. The implementation of Aquarium is characterized by pervasive data immutability, actor message passing, and service orientation.

1 Introduction

Public cloud infrastructures have emerged as an alternative to building and maintaining expensive proprietary data centers [5]. An important part of all public clouds is resource management and customer billing [1]. Even though all proprietary platforms feature mechanisms for billing customers for resource usage, there is currently a lack of open solutions.

In this paper we present Aquarium, an open source resource billing software, designed to handle the production requirements of GRNET's public Infrastructure as a Service (IaaS) platform. Aquarium utilizes a custom Domain Specific Language (DSL) for configuring the supported resources, the price lists, and the billing algorithms. It receives input from an event queue and presents billing results through a REST API. It has been developed in Scala, using the Akka library to handle concurrency and actor-based event processing. In the following sections we present the context that has shaped Aquarium's design, the software architecture and implementation of important computational algorithms, and a preliminary evaluation of Aquarium's performance.

2 Context

Aquarium is developed to provide accounting and billing to GRNET's cloud computing services—although its design ensures that it can be used by other cloud infrastructures. These services include provision of VMs and online storage; additional services will be built on top of them (for instance, repository services and services for big data computations). GRNET's services are offered to the whole Greek research and academic community, with tens of thousands of potential users. Aquarium should be therefore able to:

- Provide accounting and billing for different resources, not all of them known in advance.
- Allow the application of different pricing policies to different resources, for different users, at different periods of time.
- Allow for dynamic modification of any of the above, while maintaining full traceability of changes.
- Provide a consistent (not just eventually consistent) view of users' resource usage and billing while scaling to thousands of concurrent users.

As Aquarium must accommodate different resources with different policies for different users over time, we were faced either with a logistical nightmare of giving users resource chunks directly, or adopting a monetary approach, where users are given credit that they can use as they see fit. While in GRNET's case users are not charged real money for using the services, so the credit is virtual, this should not make any difference to Aquarium. GRNET will supply virtual currency to its users, which they will be able to spend on acquiring and using resources. If real cash were used, GRNET (or any entity using Aquarium) would only need to substitute real

```
{ "id":"4b3288b57e5c1b08a67147c495e54a68655fdab8",
  "occuredMillis":1314829876295,
  "receivedMillis":1314829876300,
  "userId": "31",
  "cliendId": "pithos-storage-service",
  "resource": "diskspace.1",
  "value": 10,
  "instance-id" : 3300
}
```

Figure 1: A JSON-formatted `ResourceEvent`

credit for virtual credit, without any changes in Aquarium itself.

Aquarium is in the critical path of user requests that modify resource state; all supported applications must query Aquarium in order to ensure that the user has enough credits to create a new resource. This means that for a large number of users (in the order of tens of thousands), Aquarium must update and maintain in a queryable form their credit status, with soft real time guarantees.

Being on the critical path also means that Aquarium must be highly resilient. If Aquarium fails, even for a short period of time, it must not loose any billing events, as this will allow users to use resources without being charged. Moreover, in case of failure, Aquarium must not corrupt any billing data, while it should reach an operating state very fast after a service restart.

3 Resources, Events, Policies and Cost Calculation

A resource represents an entity that can be charged for. Aquarium does not assume a fixed set of resource types and is extensible to any number of resources. A resource is associated with a name and a cost unit. Depending on whether a resource can have many instances per user, a resource can be complex or simple. Finally, each resource is associated with a cost calculation policy.

Usage of resources by users of external systems triggers the generation or resource events, which are received by Aquarium and charged for on a per user basis. As an example, after a successful file upload to a cloud storage service, a resource event for the diskspace resource along with the amount of bytes consumed will be sent to Aquarium. Figure 1 presents an example of a JSON-formatted resource event. Apart from resource events, Aquarium also processes events relating to users (e.g., user creation).

The cost calculation engine in Aquarium is configured by a custom DSL, based on the YAML format. The DSL enables us to specify resources, charging algorithms and price lists and combine them arbitrarily into agreements applicable to specific users, user groups or the whole system. It supports inheritance for policies, price lists and

```
resources :
  – resource :
    name: bandwidthup
    unit : MB/hr
    complex: false
    costpolicy : continuous
pricelists :
  – pricelist :
    name: default
    bandwidthup: 0.01
    effective :
      from: 0
  – pricelist :
    name: everyTue2
    overrides : default
    bandwidthup: 0.1
    effective :
      repeat :
      – start : "00␣02␣*␣*␣Tue"
        end:    "00␣02␣*␣*␣Wed"
      from: 1326041177          // Sun, 8 Jan 2012 18:46:27 EET
algorithms :
  – algorithm :
    name: default
    bandwidthup: $price times $volume
    effective :
      from: 0
agreements :
  – agreement:
    name: scaledbandwidth
    pricelist : everyTue2
    algorithm : default
```

Figure 2: A simple billing policy definition.

agreements and composition in the case of agreements. It also facilitates the definition of generic, repeatable debiting rules, that specify periodically refills of users credits.

In Figure 2, we present the definition of a simple but valid policy. Policy parsing is done top down, so the order of definition is important. The definition starts with a resource, whose name is then re-used when attaching a price list and a charging algorithm to it. In the case of price lists, we present an example of *temporal overloading*; the `everyTue2` pricelist overrides the default one, but only for all repeating time frames between every Tuesday at 02:00 and Wednesday at 02:00, starting from the timestamp indicated at the `from` field.

Cost calculation In order to charge based on the incoming resource events, time (T) and the unit of measure (U_R) for a resource (R) play a central role. Below, we present charging scenarios for three well-known resources, namely bandwidth, diskspace and vmtime. For the analysis of each case, we assume:

- The arrival of two consecutive resource events, happening at times t_0 and t_1, with a time difference of $\Delta T = t_1 - t_0$.
- The total values of resource R for times t_0 and t_1 are U_R^0 and U_R^1 respectively.

- A ratio of the form $[\frac{C}{D}]$ represents the resource-specific charging unit, where C is the credit unit, and D depends on the combination of the previously discussed dimensions (T, U_R) that enters the calculation. The meaning of the factor is "credits per D".

bandwidth In this case, an event at t_1 records a change of bandwidth, using the relevant unit of measure U_R. The credit usage computation is $\Delta U_R \cdot [\frac{C}{U_R}]$. For example, let $\Delta U_R = 10\,\text{MB}$; then the bandwidth charging unit $[\frac{C}{U_R}]$ is "*credits* per MB", since U_R is measured in MB.

diskspace We take into account the disk space U_R^0 occupied at t_0 together with the time passed, ΔT. The credit usage computation is $U_R^0 \cdot \Delta T \cdot [\frac{C}{U_R \cdot T}]$. That is, when we receive a new state change for disk space, we calculate for how long we occupied the total disk space without counting the new state change. If we had 1 GB at $t_0 = 1\,sec$ and we gained another 3.14 GB at $t_1 = 3.5\,sec$ then we are charged for the 1 GB we occupied for $3.5 - 1 = 2.5$ seconds. The disk space charging unit $[\frac{C}{U_R \cdot T}]$ is "*credits* per GB per *sec*", assuming U_R (disk space) is measured in GB and time in *sec*.

vmtime Events for VM usage come into pairs that record on and off states of the VM. We use the time difference between these events for the credit usage computation, given by $\Delta T \cdot [\frac{C}{T}]$.

The charging algorithms for the sample resources given previously motivate related cost policies, namely discrete, continuous and onoff. Resources employing the discrete cost policy are charged just like bandwidth, those employing the continuous cost policy are charged like diskspace and finally resources with a onoff cost policy are charged like vmtime. Due to space limits we omit a more detailed analysis and the description of more involved scenarios.

Scheduled tasks compute total charges, the updated resource state and a total credit amount for each billing period. This computation is recorded in a persistent store, in an append-only fashion, for future reference and as a cached value. This cached value can be handy in computations or in system crashes, to avoid recomputation based on the whole history of events.

4 Architecture

Architectural decisions Aquarium's architectural design is driven by two requirements: scaling and fault tolerance. Although initially we used a 3-tiered architecture, it quickly became clear that it would not meet our needs. Complications arose from the difficulty of describing versioned tree-based structures, such as the configuration DSL, in a relational format, and in making sure that resource events were described in an abstract way that would be adaptable to all future system expansions. Moreover, for performance reasons, Aquarium must maintain in-memory caches of computed values; for example the single query that cloud services will be asking Aquarium continually (number of remaining credits) must be answered within a few milliseconds for a large number of concurrent requests. Consequently, Aquarium's data processing architecture was based on the event sourcing pattern [3], while system state handling and processing components are modeled as collections of actors [4].

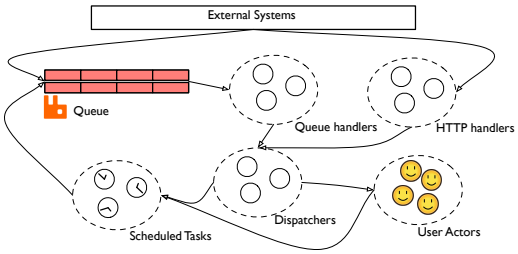

Figure 3: Functional components in Aquarium's architecture

Event sourcing assumes that all changes to application state are stored as a sequence of events, in an immutable log. With such a log, Aquarium can rebuild its state at any point in time by replaying the events in order, so it is possible to perform queries on past system states for debugging purposes. Similarly, Aquarium can concurrently employ several event processing models to cater for different front-end data requirements. Furthermore, application crashes are not destructive for Aquarium, as long as event replay is fast enough and no state is inserted to the application without being recorded to the event log first.

We use actors to encapsulate state. The actor model guarantees that only one thread touches the actor state, thus eliminating the need for locks.

Components An overview of the Aquarium architecture is presented in Figure 3. The system is modeled as a collection of logically and functionally isolated components that communicate by message passing. Within each component, a number of actors take care of concurrently processing incoming messages through a load balancer component that is the gateway to requests targeted to the component. Each component is also monitored by its own supervisor (also an actor); should an actor fail, the supervisor will automatically restart it. The architecture allows certain application paths to fail individually while the system is still responsive, while also enabling

future distribution of multiple components on clusters of machines.

The system receives input mainly from two sources: queues for resource and user events and a REST API for credits and resource state queries. The queue component reads messages from a configurable number of queues and persists them in the application's immutable log store. Both input components then forward incoming messages to a network of dispatcher handlers which do not do any processing by themselves, but know where the user actors lay. As described earlier, actual processing of billing events is done within the user actors. Finally, a separate network of actors take care of scheduling periodic tasks, such as refiling of user credits; they do so by issuing events to the appropriate queue.

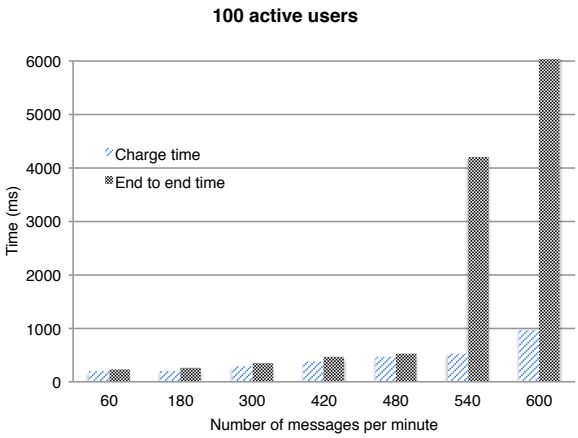

Figure 4: Average time for performing a billing operation and end for end to end message processing for 100 active users and a varying number of messages per minute.

Implementation Aquarium is being developed as a standalone service, based on the Akka library for handling actor related functionality. Akka also provided actor-based components for communicating with the message queue and, through a third party component (Spray), facilities for handling REST requests. We chose the AMQP protocol and its RabbitMQ implementation for implementing the request queue because recent versions include support for active/active cluster configurations. The persistence layer is currently implemented by MongoDB, for its replication and sharding support. However, this is not a hard requirement, as Aquarium features an abstraction layer for all database queries (currently 10 methods), which can then be implemented by any persistence system, relational or not.

5 Performance

To evaluate the performance of Aquarium, we formulated an experiment that evaluated two important properties: the time required to perform the charging operation for a resource event and the overall time required to process a resource event, end to end. To conduct the experiment, Aquarium was configured using the policy DSL to handle billing events for 5 types of resources, using 3 overloaded price lists, 2 overloaded algorithms, all of which were combined to 5 different agreements. Aquarium's data store was pre-filled in with 1,000,000 resource events, evenly distributed among 1,000 users. To drive the benchmark we used a synthetic load generator that produced random billing events, at a configurable rate per minute.

To run the benchmark we deployed Aquarium on a virtualized 4 core 2GHz class CPU and 4GB RAM Debian Linux server. The virtual machine running Aquarium was configured with a 4GB maximum heap size. RabbitMQ and MongoDB where run in another 4-core, 4GB RAM virtual machine. The two virtual machines did not share a physical host and communicated over the infrastructure's switched network fabric at an effective rate of 800 Mbits/sec, as reported by the `iperf` utility. No further optimization was performed on either back-end system.

All measurements were done using the first working version of the Aquarium deployment, so no real optimization effort has taken place. This shows in the current performance measurements, as Aquarium was not able to handle more than about 500 billing operations per second (see Figure 4). One factor that contributed to this result was the way resource state recalculations was done; in the current version, the system needs to reread parts of the event and billing state from the datastore every time a new resource event appears. This contributes to more than 50% of the time required to produce a charging event, and can be completely eliminated when proper billing snapshots are implemented. In other measurements, we also observed that the rate of garbage creation was extremely high, more that 250 MB/sec. Upon further investigation, we attributed it to the way policy timeslot applicability is calculated. Despite the high allocation rate, the JVM's garbage collector never went through a full collection cycle; when we forced one after the benchmark run was over, we observed that the actual heap memory usage was only 80MB, which amounts to less than 1 MB per user.

6 Related Work

Yousef et al. [9] described the three pricing models that are used by cloud service providers for billing used resources, namely tiered pricing, per-unit pricing and subscription-based pricing. Aquarium's cost policies that are assigned to resources map exactly to Yousef's pricing models. In fact, most offerings by public IaaS providers, including Amazon and Azure, offer services charged according to Yousef models.

Work on resource accounting and billing has been carried out in the context of cloud federation [7, 2, 6] and (earlier) grid federation projects. The Reservoir project investigated the use of service level agreements [2] for resource provisioning in federated cloud scenarios.

On the cloud computing front, vendors such as VMWare, Microsoft and IBM provide full stack solutions, which also include resource accounting. Usually, such systems are connected with existing enterprise resource planning systems. Übersmith has developed an engine dedicated to resource accounting; much like Aquarium, it tracks resource usage and applies accounting policies to it. Ruiz-Agundez et al. [8] proposed an accounting model for cloud computing based on Internet Protocol Detail Record (IPDR) and the jBilling platform. To the best of our knowledge, Aquarium is the first working open source system to offer declaratively configurable charging and accounting services for IaaS deployments.

7 Lessons Learned and Future Work

Three requirements guided our platform choices: (1) type safety, (2) concurrency using native threads, (3) distributed computation across physical CPUs. We chose Scala since the JVM had the richest collection of ready made components and the Akka library offers good scalability and distributed computation capabilities.

Regarding Scala, case classes permitted the expression of data models, including the configuration DSL, that could be easily be serialized or read back from wire formats while also promoting immutability through the use of the `copy()` constructor. The pervasive use of immutability allowed us to write strict, yet simple and concise unit tests, as the number of cases to be examined was generally low.

Akka's custom supervision hierarchies allowed us to partition the system in self-healing sub-components, each of which can fail independently of the other. For example, if the queue reader component fails due to a queue failure, Aquarium will still be accessible and responsive for the REST interface. Also, Akka allowed us to easily saturate the processing components of any system we tested Aquarium on, simply by tuning the number of threads (in I/O bound parts) and actors (in CPU bound parts) per dispatcher.

From a software engineering point of view, the current state of the project was reached using about 8 person months of effort, 2 of which were devoted to requirements elicitation, prototype building and familiarizing with the language. The source code currently consists of 7,000 lines of executable statements (including about 1,200 lines of tests), divided in about 10 packages. In the future we will add a comprehensive REST API for accessing the user actor state and we will distribute the message processing across multiple nodes in an active-active mode.

8 Acknowledgments and Availability

Aquarium is financially supported by Grant 296114,"Advanced Computing Services for the Research and Academic Community" of the Greek National Strategic Reference Framework and is available under a BSD 2-clause license from `https://code.grnet.gr/projects/aquarium`.

References

[1] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Commun. ACM 53* (Apr. 2010), 50–58.

[2] ELMROTH, E., MARQUEZ, F., HENRIKSSON, D., AND FERRERA, D. Accounting and billing for federated cloud infrastructures. In *Eighth International Conference on Grid and Cooperative Computing, 2009* (Lanzhou, Gansu, Aug 2009), IEEE, pp. 268–275.

[3] FOWLER, M. Event sourcing. Online, Dec 2005.

[4] HEWITT, C., BISHOP, P., AND STEIGER, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International joint conference on Artificial intelligence* (1973), Morgan Kaufmann Publishers Inc., pp. 235–245.

[5] LOURIDAS, P. Up in the air: Moving your applications to the cloud. *IEEE Software 27* (2010), 6–11.

[6] PIRO, R. M., GUARISE, A., AND WERBROUCK, A. Price-sensitive resource brokering with the hybrid pricing model and widely overlapping price domains: Research articles. *Concurr. Comput. : Pract. Exper. 18* (July 2006), 837–850.

[7] ROCHWERGER, B., BREITGAND, D., LEVY, E., GALIS, A., NAGIN, K., LLORENTE, I., MONTERO, R., WOLFSTHAL, Y., ELMROTH, E., CACERES, J., ET AL. The Reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development 53*, 4 (2009), 4–1.

[8] RUIZ-AGUNDEZ, I., PENYA, Y. K., AND BRINGAS, P. G. A flexible accounting model for cloud computing. In *Proceedings of the 2011 Annual SRII Global Conference* (Washington, DC, USA, 2011), SRII '11, IEEE Computer Society, pp. 277–284.

[9] YOUSEFF, L., BUTRICO, M., AND DA SILVA, D. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008* (Austin, TX, Nov 2008), IEEE, pp. 1–10.