Patterns for Scalability in the Cloud

Fabrice Troilo, Reda Bendraou
UPMC, LIP6
Paris, France
first.last@lip6.fr 2nd. author

Xavier Blanc, Jean-Rémy Falleri
Université Bordeaux 1, LaBRI
Bordeaux, France
first.last@labri.fr

ABSTRACT

Currently, interest in Cloud computing is growing. However, Software as a Service (SaaS) providers do not have any development model to manage software scalability. Without such a model, cloud software won't be able to scale efficiently and will be saturated in the case of load peak or unexpected success. We propose such a model by a mean of two cloud development patterns. Our patterns improve scalability principally by using queue to manage communications and by promoting asynchronism. Thanks to those patterns, SaaS provider will be able to develop efficient software without facing the scalability issues. We hope these patterns to be the first step of a rich collection of cloud patterns dealing with specific issues of cloud computing.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.1.3 [**Software Engineering**]: Concurrent Programming

General Terms

Cloud computing, scalability, pattern

Keywords

Cloud computing, scalability, Independent Queue pattern, SaaS, pattern

1. INTRODUCTION

With Cloud computing comes the illusion of highly scalable software [1]. To provide such an illusion to the users, the *Software as a Service* (SaaS) providers, who are the architects and developers who build Cloud softwares, have to benefit of the Cloud platform. One of the Cloud platform key principle is the dynamic creation (or deletion) of nodes. For instance in The Windows Azure Platform, a SaaS can ask at any time for more virtual machines (Azure nodes). Those new virtual machines are used to make the Cloud software scalable.

As a consequence, Cloud software is built to be deployed on a dynamic set of nodes. However, a Cloud software is composed of a finite set of components. When a component is deployed on several nodes, each copy runs orthogonally, with its own state and interactions. The main problem is then to define how each component can be deployed on the dynamic set of nodes with the intent to improve scalability by managing the set of copies and nodes.

In this main problem, we propose to identify two issues.

- The first one deals with resource containment. When a component owns a data that is shared by several other components, then its replication needs to face data inconsistency identification and resolution.
- The second one deals with coupled components. When two or more components are strongly coupled and when their life cycles are dependent, then their replication needs to face life cycle synchronization.

Even if there are probably many other issues related to this main problem, we claim that those two issues are major as they are met frequently in software design. While many research papers deal with Cloud computing definition [15, 19, 1], to our knowledge there is no approach that guide SaaS developers to improve scalability. Working for a development model in the Cloud is however an important point identified by european experts [6].

In this paper we propose two Cloud patterns. By following those patterns, SaaS providers will benefit of solutions for the resource containment and coupled component issues. This paper proceeds as follow. Section 2 introduces our two patterns, named the *Independent Queue pattern* and the *Event-Driven Independent Queue pattern*. In section 3, those patterns are used on different software in order to highlight their advantages and their limits. This section also presents a discussion about scalability challenges regarding those patterns. Finally, conclusions are presented in Section 5.

2. PATTERNS FOR SCALABILITY

The Cloud reinforces the *Service-Oriented Architecture* (SOA) design principle defining that *the more loosely coupled the components of the system, the bigger and better it scales*. The main challenge is then to be able to scale on demand. This is identified in [1] as *"the illusion of infinite*

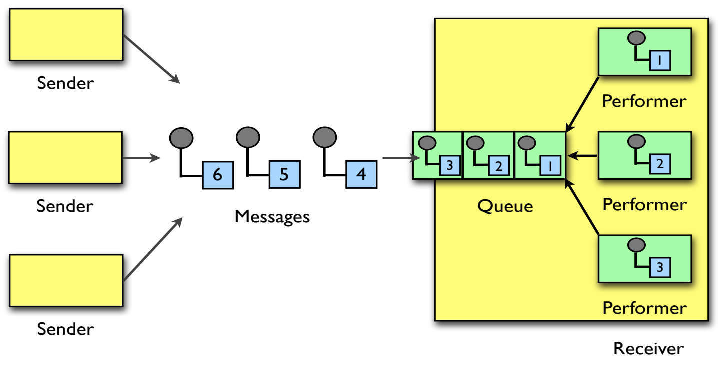

Figure 1: The Independent Queue pattern

computing resources available on demand" and in [15] as resources that *"can be dynamically reconfigured to adjust to a variable load (scale)".*

The "scale on demand" promised by the Cloud computing is strongly needed in the industry. This ability for a software to be elastic has several advantages. It allows small companies to provide new services that can accept a potentially unlimited number of client, without purchasing a data center. The companies therefore avoid risks such as success/failure by acquiring a too small/large data center. For instance, e-commerce web site has to scale automatically when new items for sale provoke a peak demand. If the developers did not consider to manage this behavior, the web site will be saturated and will miss sales.

This scaling is done by the dynamic creation or deletion of nodes. These nodes are Virtual Machine, isolated from the others. A class, a package or a module that encapsulate a set of related functions or data, which have a specific role or treatment, is what we call a **component**. Also, each component of a SaaS is deployed to an undisclosed location -but in a Virtual Machine isolated from the others- and can be replicated an unlimited number of times. Their number have to be managed proportionately to the number of clients for each component of a SaaS, which means a different approach when designing a SaaS. For instance, if a SaaS provides several payment models (such as credit card, paypal, etc.), each one should be in different nodes and should not be replicated on the same number of nodes.

To our knowledge, while many papers discusses a Cloud computing definition, a few provide tools to help developers using on demand scalability. Also, the majority of them is strongly linked with a particular supplier, such as S. Guest [11] with the Windows Azure Platform or J. Varia [17, 16] with the Amazon Web Services. Also, many designers and developers swarm of questions on the providers' forums, looking for help when developing a software with this new paradigm.

The challenges in Cloud computing stem from the fact that applications have to be scalable, but this property is not achieved just by replicating on-premise application components/nodes/virtual machines. The developers have to face this problem differently. When a component owns a data that is shared by several other components, a synchronization is needed. While several nodes are trying to access a resource for the same reason, or perhaps two different tiers want the same piece of information for two different reasons, sharing resources can be hard. SaaS providers have to take care about communication and which component have to be replicated. Also, strongly coupling components is a well known bad practice. In the cloud, a component can change location, be replaced, be removed when needed making life cycle management very complex in presence of coupled components.

In order to help Cloud designers and developers to maintain scalability in their software, we propose two patterns: the *Independent Queue pattern* and the *Event-Driven Independent Queue pattern.* These patterns help to introduce asynchrony in Cloud software, mainly by using a *First In, First Out* (FIFO) queue.

We define a **design pattern** as *"a written document that describes a general solution to a design problem that occurs commonly in many projects. A design pattern is not a finished design, and software designers have to adapt the pattern solution to their needs.".*

2.1 Independent Queue Pattern

Intent

Decouple components in order to dynamically create (or delete) each component separately. Use a queue in order to transport messages and to ensure that any component can process the messages in a FIFO order.

Motivation

This pattern is motivated by the fact that SaaS providers need to provide scalable application.

Applicability

The Independent Queue pattern is used for:

- dealing with coupled components where a component uses as input the output of an other component.
- providing scalability ”on demand” for each component.
- ensuring scalability when the number of clients is not 100% predictable.
- maintaining an asynchronous interaction between a web interface and performers.
- maintaining efficiency while replicating each components differently.

Structure

The Independent Queue pattern is referenced in Figure 1.

Participant

- A **performer** is a simple component which performs some treatment. The performers are stateless.
- A **sender** is a component that produces messages.
- A **queue** manages messages between the sender and the performers, allowing the jobs being asynchronous. The queue guarantees that no work item enqueued will ever be dequeued more than once by the performers

Consequences

If a performer is down, another one can be deployed in order to maintain the application. Each side can be replicated infinitely. Performers never manage leasing and locking, so they focus on the job to be done. The queue guarantees that no work item enqueued will ever be dequeued more than once by performers. The performers only have to get the message in the queue. The queue allows to scale simply on demand as the queue gets longer and a significant number of performer is deployed. Developers can manage this point by a script which allows to scale without looking constantly on the application.

Related Patterns

This pattern fits very closely with the Message Dispatcher pattern as identified in the book Enterprise Integration Patterns [12]. The key difference is that the dispatcher pushes messages to the consumers whereas performers will put the messages to the queue.

2.2 Event-Driven Independent Queue Pattern

Intent

Decouple components in order to dynamically create (or delete) each nodes separately. Use different queues linked to a specific event handler to perform action with a shared resource or a database.

Motivation

This pattern is motivated by the fact that a shared resource or a database can be the bottleneck of a Cloud software. In the Cloud, there is a potentially unlimited number of components that have access to a single resource.

Applicability

The Event-Driven Independent Queue pattern is used for:

- managing multiple replicates of a component or different components accessing to a single shared resource / a database.
- providing scalability ”on demand” for the components accessing to the shared resource / the database.
- ensuring scalability when the number of clients is not 100% predictable.
- maintaining an asynchronous interaction between a component and a shared resource / a database.
- maintaining efficiency while replicating each components differently.

Structure

The Event-Driven Independent Queue pattern referenced in Figure 2.

Participant

- A **sender** is a component which performs some treatment or a web interface.
- A **queue** manages messages between the sender and the performer, allowing the jobs being asynchronous. The queue guarantees that no work item enqueued will ever be dequeued more than once by performers
- An **event handler** which processes treatment in the queues.
- A **shared resource** or **database** on the Cloud which not guarantee the *Atomicity, Consistency, Isolation and Durability* (ACID) set of properties.

Consequences

The Event-Driven Independent Queue pattern has the following consequences. It allows to scale ”on demand” the sender without making the communication with the shared resource a bottleneck. It allows to build a loosely coupled event-driven system, using both queues and Cloud storage. To our knowledge, this pattern has not been associated in a Cloud environment.

Technical notes

Event-Driven Architecture (EDA) is interesting when using the Cloud and more globally when trying to loosely couple components. EDA can be defined as a software architecture pattern promoting production, detection, consumption of, and reaction to events. In a 2006 presentation [5], K. Many Chandy says that *”the business proposition of EDA is that it helps enterprises respond to events”.* In [3], Brenda M. Michelson insists on the extreme loose coupling in the EDA: *”by its nature, an event-driven architecture is extremely loosely coupled, and highly distributed”.* The sender of the event only knows the event transpired and has no knowledge of the event's subsequent processing. Thus, event-driven architectures are best used for asynchronous flows of treatments and information. For each event defined in a Cloud application, such as *ADD* and *DELETE* in a

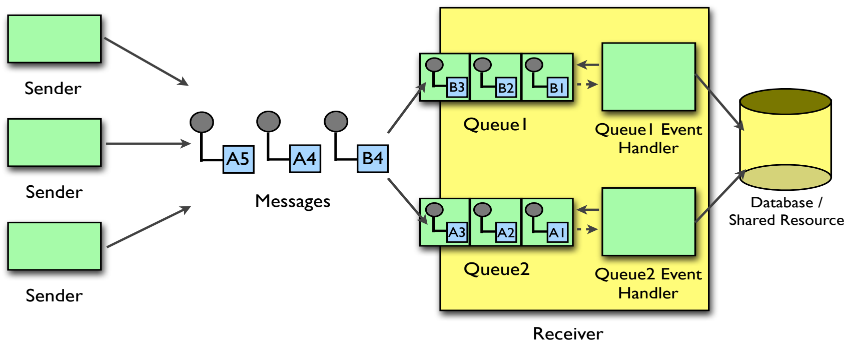

Figure 2: Event-Driven Independent Queue pattern

database, a queue is created with a specific topic and each one is linked with an event handler in the Cloud.

It is important for scalability to use a queue for each different actions. For instance, the use of distinct queues for ADD, MODIFY, DELETE actions in a file is the coarsest-grain decomposition in order to avoid a new bottleneck in your system.

3. DISCUSSION

In this section we discuss how our patterns resolve scalability on the Cloud by exposing technical challenges they resolve. The strengths of our pattern, which allow developers to easily deal with scalability in the Cloud, is to allow them to effortlessly solve important technical challenges: resource containment and loose coupling. Others benefits are identified in our validation such as the great user-perceived quality.

3.1 Technical challenges

- The first challenge is to deal with the problem of resource containment. Whether multiple nodes are trying to access a resource for the same reason, or perhaps two different tiers want the same piece of information for two different reasons, sharing resources can be hard.

  In the Event-Driven Independent Queue pattern, adding a queue between a *performer* and a *shared resource* solves this problem by promoting access to a resource using the FIFO queue. In order to build a highly scalable application, developers have to take care that our queue will not be a bottleneck in the system. The solution is to use a queue for each action in an application.

- Cloud computing applications have to be scalable, the number of each component have to to be modified on demand. High coupling does not allow it and improves dependency and scalability problems. If two components are highly linked, each one could not be duplicated on demand or replaced by a new instance in the middle of the communication. To make auto-scale possible, one component should not change or access local data of another component. For instance, using design pattern such as *Model View Controller* makes it easier to achieve loose coupling, because it makes it possible for the controller to work with multiple different Model and View components. In [13], J. Douglas Orton defines a model of loose coupling theory.

  Developers must first work on the coupling of its application. In a sense, queues used in the Independent Queue pattern do not decoupled components: their uses force developers to loose couple applications.

  Thus, developers need to ask themselves some questions, identified in [17]: *"which business component or feature could be isolated from current monolithic application and can run standalone separately? And then how can I add more instances of that component without breaking my current system and at the same time serve more users?*

3.2 Benefits

- In [2], N. Bhatti, A. Bouch and A. Kuchinsky introduce the term *User-Perceived Quality* to define the user tolerance for interface latency. Often overlooked, this part is really important for any software and especially for cloud software. The user interface has to react instantly even when a user action requires many treatments (saving data, sending e-mail, etc.). If not, the user will consider that the software has canceled his request. He will then probably venture to reload the page or press the submit button again.

  The Independent Queue pattern and the Event-Driven Independent Queue pattern increase the user-perceived quality by allowing components to interact asynchronously. Thanks to those pattern, the user interface interacts asynchronously with the components that compose the cloud software. Moreover, solutions such as Ajax [9] may be used to notify to the user that his request is processed.

- Replicating a database performer is always a challenging task, especially for the cloud domain. This has to deal with concurrent access, locking mechanisms and consistency.

  Thanks to our two patterns, whatever the number of copies, the complexity for managing components is always the same.

3.3 Future works

In [4], E. Brewer[1] states that it is impossible for a web service to provide the following three guarantees :

- Consistency : there must exist a total order on all operations
- Availability : the system have to be available 24x7, despite transient partial hardware or software failures.
- Partition-tolerance : the system must work, even if at a given time, two partitions in the networks are done and can't refresh each-other.

This result, also knows as the CAP theorem, has been proved by S.Gilbert and N.Lynch [10]. Authors have shown that, for any web service, it is only possible to achieve **any two of the three properties**. In the Cloud emphasizes for partition-tolerance and 24x7 availability in order to make system scalable and available. It then appears that it is not feasible to guarantee strong consistency, hence the introduction of more flexible system consistency [18] like BASE, a *weaker-than-ACID data semantics* [8, 14].

The two patterns guarantee consistency for simple transactions, such as incrementing a shared resource or saving changes in a specific file or tuple. By using one FIFO queue for each resource or database file in the Event-Driven Independent Queue pattern, the Event Handlers associated guarantee consistency by working with the queues elements one by one in the FIFO order.

As E. Brewer's proof assures that it not feasible to guarantee strong consistency with partition-tolerance and 24x7 availability. As a major limit, our approach does not guarantee consistency for complexes transactions concerning multiple files, without using a unique non-replicable component which manages locking and unlocking data. Future works will address lower consistency in the Cloud environment.

4. VALIDATION

In order to use these patterns and to validate their utility in a Cloud development, we have developed different software.

4.1 CAFE

The first software is *A Collaborative javA File Editor*[2] (CAFE). The principle is to propose a file editor on the web, with actions similar to *SubVersion* or *GIT*. Each user can upload and edit java files (stored in *Blobs*) in the Cloud via a Web Browser. Blobs are reachable via REST (HTTP GET, POST, PUT, DELETE) at a specific URL. All other users have access to a file once it has been uploaded, so they can edit and modify it. Each user has access to a personal version of the files edited by its own, and a *"repository"* version of each file is maintained.

In CAFE, we used the Independent Queue pattern and the Event-Driven Independent Queue pattern in order to manage scalability.

The Independent Queue pattern is used to maintain asynchrony between the web interfaces and the performers, which manage actions such as DIFF between two files. For instance, when a user wants to save a file, a case is added in the queue 1 between the web interface and the performers. This case contain the content of the file and other information used to locate the user. This allows to manage asynchrony between this two components and to inherit all the benefits mentioned above. If the queue 1 grows and exceeds a number of 10 cases, a script automatically deploy a new performer component.

We use the Event-Driven Independent Queue pattern for storing changes in the Blobs. When a performer has read in the queue 1, it has to work with it. The others queues are linked with event handlers, managing commit, update, etc;

In Windows Azure, the queue component is implemented with the architectural style REST [7], which is a good choice for performance and simplicity. Interactions with the queue are basics with the HTTP verbs GET, PUT, POST and DELETE.

4.2 The Parking Problem

We have implemented a *Parking Problem.*

It represents a parking, in which the number of spot has to be scalable. We identify the 4 following components:

- A web UI is used to add manually a new car.
- The queue which represent the cars.
- The performer which represent the gates.
- A blob which represent the number of spot remaining in the parking.

When the user wants to represent a new car arriving at the parking, he presses the submit button to add a new car to the queue. Then, the car is added in the unique queue for process, by following the principle used in the Independent Queue pattern. The performers which represent the gates, manage the treatments with the queue. The number of spots remaining is stored in a REST database. If the number of spot is positive, the performer can read in the queue in order to simulate the entry of a car in the parking. The properties of the queue ensure that a car can be brought in just one spot. The benefits is that the number of gates can be increased or decreased on demand. Each gate, when ready to bring in a car, reads in the queue. For this shared resource problem the queue has shown to be really useful, and easily understandable for developers.

5. CONCLUSION

In this paper, we have focused on the scalability behavior in Cloud computing applications. We believe that the key

[1]Eric Brewer is a professor at the University of California, Berkeley, and the co-founder and Chief Scientist of Inktomi
[2]This application has been implemented by using the Windows Azure Platform

point is to maintain queues between components. They allow to scale on demand as they get longer. Moreover, we introduced two patterns for developers who need a development model for working in the Cloud, the Independent Queue pattern and the Event-Driven Independent Queue pattern. By managing queues, these patterns allow to scale on demand, and help developers to maintain asynchrony between the components. They then improve the user-perceived quality.

The major contribution of the asynchrony is to encourage developers to decouple the components and to work in a stateless model. This allows to duplicate components on demand safely.

Few recent on-line articles attempt to establish frameworks, APIs and patterns for Cloud computing. Although ideas are interesting, they are sometimes complexes or linked to a specific Cloud provider such as Amazon Web Services or the Windows Azure Platform. To our knowledge, there is no research papers which introduce patterns such as ours.

Because Cloud computing is a young paradigm, many papers are still dealing with the Cloud computing definition, and very few are searching for a development model. Though, developers need patterns level for application scalable such those we provide in our paper. We believe our patterns comprehensive and easy to use for a developer who started in the Cloud.

6. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[2] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 1–16, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[3] M. M. Brenda. Event-driven architecture overview. *Patricia Seybold Group*, pages 10–1571, 2006.

[4] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM.

[5] M. K. Chandy. Event-driven applications: Costs, benefits and design approaches, 2006.

[6] K. J. [Ercim] and B. N.-L. S. Research], editors. *The Future Of Cloud Computing, Opportunities for European Cloud Computing Beyond 2010*. EUROPA > CORDIS > FP7, January 2010.

[7] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Taylor, Richard N.

[8] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev.*, 31(5):78–91, 1997.

[9] J. J. Garrett. Ajax: A new approach to web applications, February 2005. [Online; Stand 18.03.2008].

[10] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[11] S. Guest. Patterns for cloud computing, 2010.

[12] G. Hohpe and B. Woolf. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, October 2003.

[13] D. J. Orton and K. E. Weick. Loosely coupled systems: A reconceptualization. *The Academy of Management Review*, 15(2):203–223, 1990.

[14] D. Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[15] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[16] J. Varia. Cloud architectures. *White Paper of Amazon*, 2008.

[17] J. Varia. Architecting for the cloud: Best practices, 2010.

[18] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[19] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov 2008.