Scaling an AWS infrastructure - Tools and Patterns

Monday, August 16, 2010 at 7:52AM

Todd Hoff in AWS, Strategy

This is a guest post by Frédéric Faure (architect at Ysance), you can follow him on twitter.

How do you scale an AWS (Amazon Web Services) infrastructure? This article will give you a detailed reply in two parts: the tools you can use to make the most of Amazon's dynamic approach, and the architectural model you should adopt for a scalable infrastructure.

I base my report on my experience gained in several AWS production projects in casual gaming (Facebook), e-commerce infrastructures and within the mainstream GIS (Geographic Information System). It's true that my experience in gaming (IsCool, The Game) is currently the most representative in terms of scalability, due to the number of users (over 800 thousand DAU – daily active users – at peak usage and over 20 million page views every day), however my experiences in e-commerce and GIS (currently underway) provide a different view of scalability, taking into account the various problems of availability and data management. I will therefore attempt to provide a detailed overview of the factors to take into account in order to optimise the dynamic nature of an infrastructure constructed in a Cloud Computing environment, and in this case, in the AWS environment.

The Tools

I will provide examples of tools I used in my various production experiments and which won me over with their efficiency. However, it's not so much the tools themselves which are emphasised, rather the underlying features that must be installed to fully exploit the AWS.

What is the difference between an AWS and a standard infrastructure?
There are two replies:

- Firstly, the AWS enable the Hardware and Infrastructure components to be handled via the code (through the APIs): this gives them momentum which needs to be intensified by correctly selecting your tools. Amazon also offers a certain number of technical guarantees with their services, which should be used to full advantage in order to concentrate on what is essential.
- The second answer should be: 'None!' However, it must be acknowledged that often the cosy aspects of 'home'-hosted infrastructure can lead to a certain lack of rigor on many issues. The fact that Amazon, with its AWS, offers a dynamic and volatile solution for the EC2s, compels you to install mechanisms which should be standard.

Centralised configuration management
In order to instantiate multiple servers (EC2) on the fly to counter a load peak or perform occasional procedures, we have to use a centralised configuration management tool. This has multiple uses:

- Rapid instantiation of a machine corresponding to a pre-defined type: Web server, cache server, etc.

- Ensuring uniform configuration of the entire set of instances of the same type at all times.
- Capitalising on the expertise: packages, configurations, services etc. for each type of instance are contained within a descriptor. This enables new users (and others) to learn the composition of the servers simply by reading the associated descriptor(s.)

Puppet is a very attractive solution for this task. Puppet architecture comprises:

- A Puppet Master, guardian of configurations. It contains the descriptors of the packages to be installed, configuration files to be pushed and services to be started up for a new EC2 instance in order to prepare it according to the type of node or nodes with which you have associated it, or simply to keep an existing instance updated.
- Multiple Puppet clients installed on EC2 instances. The client regularly polls the master to check that his configuration is up to date. It is also possible to trigger the clients from the master to initiate an urgent update.

It is thus very simple to mount a new instance of a given type, and to be sure that from one server to another, the configuration is identical in every way. It should be noted that the Puppet Master descriptors run on a node system, and a server may reference more than one node. You will therefore obtain a very modular configuration.

```
class snmp {
      package {
            « snmpd» :
            ensure => installed;
      }

      file { « /etc/default/snmpd» :
            ensure => present,
            owner => root,
            group => root,
            mode => 0644,
            source => « puppet:///snmp/snmpd» ,
            require => Package["snmpd"],
            notify => Service["snmpd"];
      }

      file { « /etc/snmp/snmpd.conf» :
            ensure => present,
            owner => root,
            group => root,
            mode => 0644,
            source => « puppet:///snmp/snmpd.conf» ,
            require => Package["snmpd"],
            notify => Service["snmpd"];
      }

      service { « snmpd» :
            ensure => true,
            hasrestart => true,
            restart => « /etc/init.d/snmpd restart» ,
            hasstatus => true;
      }
}
```

Puppet Module Descriptor

The centralised configuration manager is indispensable on a scalable infrastructure, which can have a large number of instances, some of which should be mounted rapidly, in order to reply to user requests during load peak.

‘But’, you may say, ‘you *do* have to install the client on the new instance, don’t you?’ Oh yes, all the more so as there is a system of certificate request/acceptation between the client and the master, so that not just anybody can access the configurations. ‘There’s nothing simpler!’ I reply. Instead of using an HMI (Human

Machine Interface) such as Elastic Fox to connect to the Amazon services APIs, you need simply to connect with command lines via a script which instantiates an EC2, creates if necessary an EBS volume and associates it, connects with SSH on the new instance and installs the client, connects on the master and accepts the certificate request, reboots the client and 'Bingo!' The instance installs and configures itself. In general, the reactive and dynamic capacities inherent in EC2 and AWS require it, otherwise it would be an under-exploitation, or even a misuse of the tool.

And what about the AMIs (instance-store root devices) or the EBS root devices (note that for the latter there is not yet a great choice of templates)? It is however possible to build an AMI or even an EBS root device with the Puppet client installed on it (and to use, if necessary, the user-data at the launch of the instance in order to set the parameters if a 'static' configuration on the AMI is not sufficient). However, in a production environment, the AMI and the EBS root device do not allow you to replace a Puppet: if you have to modify the AMI every time you change the parameters and redeploy the EC2… you'll never finish. The AMI is a good tool for instantiating a basic template, but not for maintaining a production infrastructure in good working order.

To sum up, it is even possible to envisage auto-scalability (a more complete service than the one offered by Amazon's Auto Scaling, based on CloudWatch), which would amount to starting up this script when a value on a monitoring graph (metrology) exceeds a threshold, thereby instantiating a new instance to weather the peak period, an instance which would be 'terminated' after dipping below another threshold on the aforementioned graph. The possibilities are enormous. However, the cost of the final elements to be installed on a large production infrastructure to access the auto-scalability function is considerable (exponential complexity). If you get the chance, it's an excellent handiwork, otherwise you'll have to 'settle' for an easily scalable solution whereby you simply launch a script manually (or else at a fixed time based on the study of the metrics produced by the metrology, which is a satisfactory alternative). If necessary, manually adding one or two references in a file is a perfectly reasonable alternative.

Execution of automated tasks in parallel

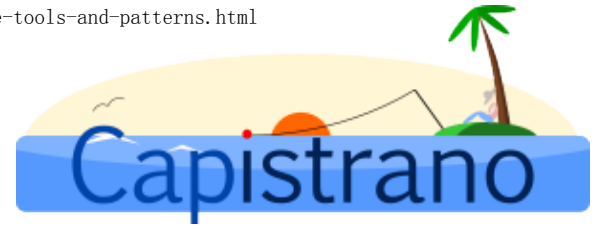
Because of the variation in the number of instances according to the load of a scalable infrastructure, and particularly in the potentially large number of these instances, you cannot envisage carrying out maintenance and deployment operations on an individual basis for each machine (or manually, which is even worse!). This would not only be too time-consuming, but would also lead to errors. It is therefore essential to use tools specialising in automation and parallel task execution, such as Capistrano, which has numerous advantages:

- Scripting a certain number of tasks, whether complex or not (deliveries, backups, site publication/ maintenance, etc.) executing them rapidly in parallel on X instances with a single command.
- Ensuring the reproducibility of a task.
- Capitalise on know-how (Cf. Puppet).

```
task :dump_sql, :roles => :sql do
      run « rm -f /mnt/backup/`hostname`.`date +%w`-*»
      run « mysqldump -pmon_pwd -u mon_user ma_base | gzip -cq6 > /mnt/backup/`hostname`.`date +%w-%y%m%d_%H%M`.sql.gz»
end
```

This tool is very practical and easy to use. It executes tasks in parallel (by connecting with SSH – don't use the 'root' key of your EC2 instances in the tool, it's classier to use a dedicated

key…) on one or a group of servers defined by roles.



This tool should be used as a supplement to Puppet and not a replacement, as they do not have the same targets:

Objective
Puppet - Uniform configuration of the server pool.
Capistrano – Task reproducibility and in-parallel execution.

Operating mode
Puppet – Regular pull requests by clients (or even push from the master).
Capistrano - Occasional tasks (manual request or by cron).

Orientation
Puppet - Pre-defined objects / notions such as ‘Package’, ‘Service’, ‘File’, etc.
Capistrano – Generic commands such as ‘upload’, ‘download’, ‘system’, ‘run’, etc.

Target
Puppet - Infrastructure and services.
Capistrano - Services and applications.

Note that Webistrano is an HMI enabling access to the Capistrano features and introducing the notion of project management by managing access to the tasks according to profile, to trace who deployed what on which server and to send email signals in response to certain actions. I find this joint operation with project management very interesting.

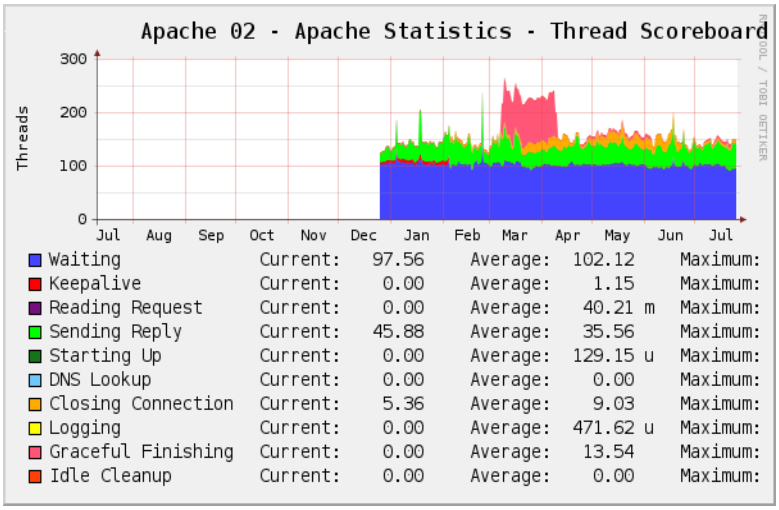
Monitoring
This is one of the crucial elements of a scalable architecture, if not the main one. It is essential to have the metrics to hand, precisely so you know when to scale. This enables you to identify the clogged points of the infrastructure, to analyse your site traffic and to make deductions about user behaviour. It also enables signals to be triggered.

The provision of metrics is more representative of metrology than of supervision. We will see that, in the context of an AWS infrastructure, metrology has greater added value than supervision (even if the latter remains indispensable). Speaking of which, there follows a clarification of the two monitoring disciplines that you need to recognise:

- **Supervision** verifies the state of a host or a service and sends out an alarm upon detecting any abnormal behaviour (over-long response time, status NOK, etc.) and requires immediate action on the part of the interface partner concerned. An alarm must signify that the host or the service is unusable (critical) or may soon be (warning) and must not send back ‘simple information’ which would obstruct the visibility of real incidents.
- **Metrology** enables data to be archived, and if necessary, processed or filtered, before it is presented in the form of graphs or reports. This data enables corrections to service development or configuration to be carried out after the event, in order to optimise the aforementioned services, and equally to define the resources to be attributed to them in the most effective way. This aspect of monitoring is just as important, for it will enable the service to be improved (and thereby the users feedback) and also to

reduce costs. The metrology results should clearly highlight the improvements to be made by correlating the values recovered.

Quite a number of monitoring tools are available on the market, each with its pros and cons, some more supervision-oriented, others more towards metrology. Among them: Centreon/Nagios, Zabbix, Cacti and Munin. Personally I use Centreon/Nagios mainly for supervision and Cacti for software-oriented metrology with pre-packaged graphs such as for Apache, MySQL, Memcached, etc. I tried Munin, it lacks some of the useful features of Cacti but it is really easy to use and it matchs very well with Puppet (configuration based on descriptors and symlinks). I've also heard many good things about Zabbix (comprehensive range of features).



Thread Scoreboard Apache - Yearly - 1 Day Average

It should be noted that these tools operate mostly (Centreon, Cacti and Munin) on RRDtool (Round-Robin Database), a database management tool which also enables a graphic representation of the data contained in the base. The big advantage stems from the fact that the stored data volume is minimal, whether for storage of several months, or even years. This is made possible by calculating the averages of the time periods (from 1 minute to 1 day): the recent data is exact, whereas the older data is approximate. This is very practical, as you have both the recent exact metrics and you keep the historic, a representation of the evolution of your architecture. This is excellent for visualising the evolution of the key metrics and understanding the impact of the evolution of the components of the aforementioned infrastructure, or the applications supported as and when the different releases are delivered.

As for Cloud Computing-type infrastructures, metrology is the discipline with the highest added value, as it will enable feedback to your automation tools.

Keep in mind also that it's interesting to be able to integrate dynamically the new instances installed and configured by Puppet in the monitoring tool. Give priority therefore to monitoring tools with modular configuration or simple access via APIs.

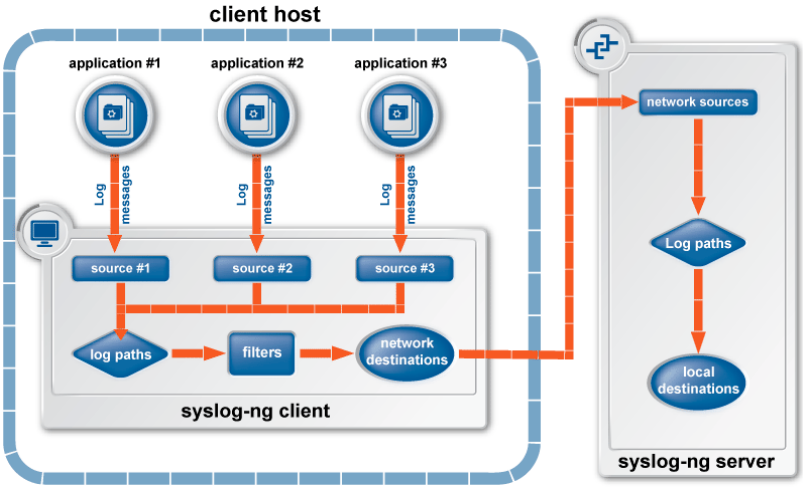
Managing centralised logs

It's undeniable: the more instances you have, the more scattered logs there'll be on the various instances. Moreover, sending all the logs to the syslog daemon becomes difficult, as the infrastructure components do not necessarily manage it very well (redirection, respecting the syslog protocol, etc.), not to mention the application(s) / functional block(s) which can log in the various files on the server.

syslog-ng

Consider trying a tool such as Syslog-NG: it comprises a server and several clients, in fact it's the same tool but with different configurations. What's more, it is capable of regularly checking the log files (of business applications for example) and transmitting only the differential, even if it doesn't respect the syslog protocol:

- The client recovers the files/pipe/unix-dgram/etc. logs and sends the information to the network (TCP/UDP).
- The server recovers the information from the network (TCP/UDP) and registers it in files, databases, etc.



Syslog-NG Architecture

We're talking about only minimal differences in terms of configuration. The component can also act as a relay on more complex infrastructures by receiving the network logs from diverse clients and sending them back to the network for a server.

It is possible to apply filters before sending the logs to the network in order to send only the essential ones. Next, filters can also be applied at the server level in order to manage differently the logs coming from clients, according to their urgency, sender program, source host, etc.

It's a simple and non-intrusive tool able to take into account the logs of scattered applications on a server (even if these logs do not respect the syslog protocol format – there is however an advantage to respecting this protocol: greater accuracy of the facilities and priorities).

Keep in mind, the same as for monitoring, that it is interesting to be able to integrate (and withdraw) dynamically the new instances to be managed at the logs system level.

Conclusion

Over and above the tools presented here, it is the underlying functions and capacities which are important. What stands out are the following:

- Tools such as centralised configuration managers, tools for the execution in parallel of automated tasks and also log managers, will become even more important. Working without them on such

infrastructures is no longer conceivable.

- Monitoring is still and will remain a key value, but metrology allows you, with the analysis of key defined metrics in your infrastructure, to sustain the automation tools.
- Automation is indeed the key word promoted by AWS: the accessibility of the Hardware and Infrastructure resources via code allows us to move towards ever more autonomous architectures. However you must always position the cursor correctly, in the knowledge that the final stages of complete automation are the most expensive.

The Pattern

In this part of the article I describe the architecture model and the underlying technical components you should use in order to implement a scalable infrastructure. We will look in particular at the optimisation of data access in scale-out-type architectures suitable for implementation as a distributed system, as much at the data model level as the lower layers for I/O optimisation. We will also examine the recommended development concepts such as Stateless, in the finest REST tradition. I will end the article with some tips and tricks. My aim is to help you set up and optimise your infrastructure by understanding how Amazon tools operate and to get the most benefit from them.

Scale-out

Scale-out is the opposite of scale-up: the latter means the dynamic expansion on the fly of a given server's resources (addition of RAM or CPU). Thus, the application continues running on a single machine whose capacity has been increased. Scale-out (the AWS way), on the other hand, offers to increase the number of servers in order to respond to the increased load of an application. Ultimately, the global potential of the infrastructure increases too, but is divided among several servers. This implies that the application supported on the server had been thought out in terms of distribution, that is, it can be executed in parallel on several servers without the risk of corrupting the business logic (think to shared session management or no session, for example) or creating a problem of access to the resources (which must then be available via the network and not locally on any given server, for example).

Scale-out can be likened to Agent Smith in *Matrix*. As for scale-up, think more along the lines of the Stay Puft Marshmallow Man in *Ghostbusters*!

Stateless

The best way to obtain an application that is both easy to distribute and high performing, is to plan for it in stateless mode. This means that it's the client request which must contain all the information enabling the request to be processed by the server, either via a cookie, or else via the URL parameters. Whenever possible you should choose this approach at any price, compared to the stateful mode in which the server must keep a context for each client to be able to respond to the request. The stateless concept is one of the great REST concepts. While we're on the subject, I strongly recommend you read this excellent article on REST (in French) by Jean-Paul Figer to appreciate the style in all its simplicity and power.

If you have to manage stateful, you will need to use a network cache such as Memcached, to share the contexts among your different servers. Don't even consider the local cache and sticky session package, as you will have trouble getting even distribution of the load balancing. It's not a good idea to use a cluster which communicates in multicast either, because the Amazon network does not allow that … Or all the servers

could communicate in unicast with a gateway which takes care of redistributing the communications… Are you sure you don't prefer stateless?

The implementation of stateless applications did not arrive with Cloud Computing and AWS: on the other hand, if you don't think along these lines, you won't be able to make the most of all the energy and flexibility they provide.

Optimisation of data access
Optimising data access requires two elements:

- The data model and software architecture that are found in scalable architecture, whether you are on an AWS infrastructure or not.
- I/O management, which can be optimised in particular by thoroughly exploiting AWS.

The scalable model
This subject deserves an entire article of its own, but the scalable model is in keeping with a rather classic model within the casual gaming-type applications, or more generally linked to applications based on the social graphs, or centred on a distinct business concept in particular.

To summarize, two methods of data storage are:

- Storage of what we'll call the meta-model, which is not intended for distribution, built on a structured, indexed relational base (even though essentially used in the form of a key-value access), containing the general information of each user (in the case of a social application the following: *name*, *top score*, *previous day's wins*, etc.), primarily accessed in read and for which it is possible to alleviate the load via a Memcached in front of it. You can therefore carry out SQL-type set-based requests (ensemblist) and thus recover information by using clauses (*WHERE*). This base can be a MySQL or a PgSQL for example. I suggest you read this very interesting report by Guillaume Plessis, comparing MySQL installed on an EC2 with the Amazon RDS (Relational Database Service). I don't risk ruining the suspense by telling you that databases are a real bearded geek affair! :o)
- Storage for more volatile data (such as game data etc.), with a heavy write:read ratio, difficult to cache, and for which you must choose a real, structured, non-relational storage solution of a key-value-type (NoSQL / Not only SQL, some would say), thus enabling easy distribution of the data on X servers. I can mention a few names: Redis, Tokyo Tyrant/Tokyo Cabinet, MemcacheDB. And also the new peer to peer models such as Cassandra.

Tokyo Cabinet

In this case, Tokyo Tyrant / Tokyo Cabinet is a solution that I have implemented, which I find satisfactory

(note that a new tool Kyoto Cabinet has come and one of his worthy features is to optimize the management of multi threading and thread concurrency compared to Tokyo Cabinet). Tokyo Tyrant / Tokyo Cabinet is a twosome, managing the requests from distant servers (network interface) and access to the storage system respectively. Six APIs are available for storing the data:

1. On memory Hash,
2. On memory B+tree,
3. File Hash,
4. File B+tree,
5. Fixed-length Array,
6. Table.

Each of them has its own characteristics and responds to particular requirements in terms of performance and practicality.

Regarding performance: apart from its natural capabilities, it has the notable advantage of not managing authentication (which is usual in key-value storage systems – there are exceptions like SimpleDB, for example – security is supposed to be guaranteed at the network level for accessing the storage system and at the applications level for access to data – so that you can only retrieve that which concerns you). This reduces the access rate (no need for authentication requests) compared to a classic database. Furthermore, no reconstruction of onerous index management is required (possible however on the Tokyo Team's API 'Table') … It goes without saying that what's really time-consuming about data access via the network (between the web or application server and Tokyo Tyrant) is… the network! So in all cases, at the client (API) level you should use 'mget' (multi get), for example, by positioning a array of keys that you wish to recover in the parameters rather than performing a 'get' N times from the client, therefore creating N network accesses. 'mget' will be resolved at the Tokyo Tyrant level.

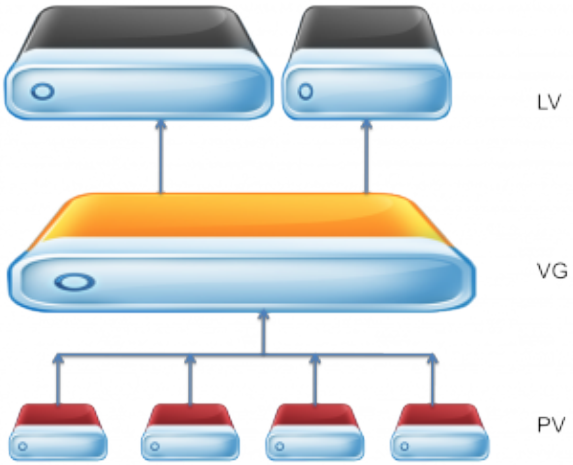
The arguments concerning this tool should be applied to all key-value-type storage.

I/O management

Now we come to the specific characteristics of AWS in terms of storage: EBS (Elastic Block Stores). EBS volumes are network disks optimised for I/Os, easy to use, ensuring the survival of crucial data during an EC2 (volatile) shutdown. On a purely practical note, I did however notice latency variances: not enormous, but on an application that is much in demand, it does get noticed. This is not insurmountable, it's simply something to bear in mind. If your servers' Load Average increases, it's not necessarily the CPU which is to blame … Think I/Os!

Firstly think striping, therefore RAID0. No need for mirroring (RAID1), as the data safety in EBS is already guaranteed by Amazon, transparently, by network replication: let's make the most of this service and concentrate on striping. Remember too to spread the writes of the different physical files (data, update logs, etc.) among different disks (for tools using update logs – and this is the case for MySQL and Tokyo Tyrant for example – do remember to set their capacity parameters). This is what allows you the best optimisations. Next, choose your filesystem carefully with regard to the tool (EXT3, XFS, etc.) and consider the mounting options (noatime, nodiratime, etc.) Finally, check the default scheduler on the EC2 instances that you are starting up: it's the NOOP scheduler by default (I/O requests in a simple FIFO file). Consider something more efficient, such as CFQ (Completely Fair Queuing). Well, I say that, but it always depends on the top software layer, and if and how it manages the priorities: in this example (which is not on EBS) on the

MySQL Performance Blog, it appears that the NOOP and DEADLINE schedulers are the best for improving the InnoDB I/O. It just goes to show, it's always better to test first for your own particular case!



Logical Volume Management

Another tool I have used with the EBS and which works very well is LVM2 (Logical Volume Management, version 2). I haven't used it with an architecture with a high number of IOPS (I/O Operations Per Second) and have therefore not tested the striping possibilities (RAID0) offered by the tool. On the other hand, I have used it with an infrastructure which had to be able to tolerate considerable increases in data volume but without interruptions in service. The snapshot and volume re-creation operations of an EBS are too long. LVM provides the solution, because this abstraction layer enables the management of software volumes independent of physical resources: you only need to raise a new EBS resource (a command line), associate it with the EC2 instance (another command line) and add this resource to the LVM resource pool. All that's needed is to increase the logical volume (still transparent, without service interruption), then extend the file system. That last operation (which is a joint operation with the re-creation of the EBS volume) can necessitate shutting down the service for a few moments (around ten seconds). Note that it is possible to hot-extend your file system with EXT3, personally, I prefer a service interruption of a few seconds on this type of very short but crucial operation. And what's more, it is easy to perform backups on an LVM system offering a snapshot differential option, which you can mount like an independent logical volume: when there is a modification to the origin volume, the initial value is copied in the snapshot volume, so you can snapshot large volumes on a limited space because only the frequency of modification is what counts. You can find further information on LVM2 and EBS by consulting this very good article (in French) by Laurent Roux.

All these examples demonstrate that EBS are a very useful resource: you have to consider them firstly as a means of ensuring the durability of important resources before taking into account the performance aspect. In fact, performance is not necessarily better on an EBS than on the local disks of an EC2 instance (Cf. a simple and interesting test (in French) by Charles-Christian Croix. Or, to see some slightly more complex tests, Google *"performance ephemeral disk ebs volume raid"*. There are quite a few very interesting cases, and I must confess I find it hard to choose between them). Nevertheless, when you are committed to ensuring the safety of your data and thus using EBS (which, for those in the know, is comparable to a LUN on a SAN), there are several ways of maximising their performance.

It should be noted that, for a pure performance solution, use them to spread the files (and thus the I/Os)

among several EBS disks.

Note however that the EC2 bandwidth is inevitably limiting, once you reach a certain threshold: adding EBS is helpful for smoothing out the variance differentials (in RAID0) or for optimising the throughput, but the fact remains that if you add several EBS to your EC2 instance and they are in great demand… it's the EC2 bandwidth that jams. You will have to add a new EC2 (scale-out to increase the resources – in this case the bandwidth), attach new EBS to it (don't forget that an EBS may be attached only to a single EC2 at a time) and partition the data onto it (sharding).

In all cases, handling the EBS is easy and this flexibility is a plus.

Tips & tricks
This last section contains some useful hints.

Aliases & IPs
Remember, on an AWS infrastructure, to position aliases corresponding to your instances' private IPs in the 'hosts' file, which you then deploy via Puppet for greater reactivity, given that the IPs are 'variable' (when you shut down one instance and start up another straight away, you will not keep the same IP). By using the the 'hosts' file you will avoid provoking internal DNS resolutions for Amazon.

CDN & S3 Headers
For the CDNs (Content Delivery Networks) that you will have to use in a scalable architecture, remember to use S3 in order to provide images and other static content. It's useless to keep a Web server only for that purpose: that's what S3 is for, and at minimal cost.

Remember also that in certain cases where your traffic is more moderate and you don't wish to change over to a CDN (for reasons of cost, for example), you can always optimise the management of your resources on S3 by using the metadata attributed to the said resources on S3, by positioning, for example, *Cache-Control* or *Expires-type* headers. You may visualise the metadata via the new S3 tab of the Amazon management console.

Backups… What else?
S3 is also the solution for backups. I have not found any integrated tools satisfactory for this task: I simply use a command line tool "s3cmd" in order to store my backups. I integrated this tool into Capistrano tasks called by cron. s3cmd is very simple to operate, enables the use of S3 services and offers the possibility of transferring data in HTTPS, and also to store it in encrypted format.

Conclusion
There are many more things to be said on this subject, but the abovementioned points are, in my opinion, the essential ones, in any case with regard to using the services provided by AWS. To sum up:

- Stateless and REST are, generally speaking, better adapted to AWS, since they enable easy sharing of the load on an infrastructure which can adapt to the increased load on a scale-out model.
- Data access always creates a bottleneck in distributed scaling applications. At this level it's a matter of working on the data model and making good use of the adequate storage systems:
  - SQL technologies for metadata able to necessitate set-based querying (ensemblist) and relational things, while making maximum use of a network cache for information with a high read:write

ratio.
    - NoSQL / Not only SQL / key-value technologies for information which can be accessed with a single key, and thereby benefit from the distribution inherent in scale-out.
- Finally, remember the lower layers by selecting an appropriate filesystem and scheduler. You should also consider RAID (striping) and LVM (volume extension and snapshot differential) to get the most out of the EBS' versatility and allocation of the I/Os on different devices (always being careful with your EC2 bandwidth, which can become restrictive: in that case add one or more EC2 instances to gain bandwidth and partition – shard – the data on new EBS).

I hope you enjoyed my report on the scalability of AWS infrastructures.
