White Paper

Design Principles for Applications in the Clouds – Why Sizing Application Entities to Platform Capabilities Matters When You Need to Scale

Abstract

Internet-scale applications utilizing cloud computing infrastructure demand that their architects achieve elastic scaling by stitching together a large set of computation entities.

This concept of pulling together multiple platforms to achieve a goal is not new – HPC-type applications have for decades achieved parallel performance by techniques such as data decomposition, where the decomposed datasets fit comfortably on each node of their compute grid. What is new with cloud computing is the need to get this ability from less well-decomposed applications. But numerous recent examples, e.g., the Google* File Systems "cell" concept, database "shards", etc., show that it is possible, as long as the designer adheres to certain simple concepts.

What is not so well understood is that, at the core, each of the applications parts, which we will call "entities" in this note, must fit "comfortably" on real hardware. This means that those applications need to effectively utilize the capability of each virtual platform that make up their compute infrastructure and not overload those platforms to the point that they can no longer compute effectively on real-world hardware. We will examine the implications of this fact in this short note.

Scaling & Entities

Let's examine the dictionary definition of scaling. That says that scaling is the ability of a system to resize itself, to make it larger or smaller based on demand. With regard to increases, "scale vertically" or "scale up" refers to expanding a single machine's capability. To "scale horizontally" or "scale out" refers to adding more machines.

Delivering "Internet Scale" brings new "scale out" architecture principles for developers. Counterintuitive to the conventional wisdom these Clouds relegate single platform's computation capacity to the background, these principles actually suggest that architects need to modularize application functionality and data and fit these into "entities" --- and these "entities" need to map 1-to-1 to physical platforms (or many-to-1 in for scale-up infrastructure which consolidates virtual machines onto large symmetric multiprocessing platforms)!

Note that we are not saying that these mappings should be static – rather they should take into account the rapid progress that hardware makes the lifetime of any software system. The highest benefit from cloud-friendly application architecture is when this mapping is modularized & can react dynamically to load; however, developers must still view single platform capabilities as tremendously important.

Some Detailed Examples

First, take the story of Animoto* and their use of RightScale control technology running over Amazon* Web Services to provide a Facebook* application which easily builds videos from a user's pictures and audio. In 2008, Animoto supported a viral 3 day growth of their Facebook application from 25K users to 250K users, a 100X increase. Think of that as going from preparing your family dinner in your kitchen on Wednesday to cooking dinner for your entire neighborhood on Saturday -- would your kitchen or your preparation techniques handle that kind of increase?

From the RightScale blog, their architecture was designed to connect all the operations using queues, many of them in AWS's SQS. One queue contains work items that list photo URLs to fetch from other sites, such as Facebook, Flickr*, etc., and that are processed by one array of worker instances. Another queue has the list of render jobs and each work item in there points to the set of photos sitting at the ready in S3 and at the music files also on S3. All of these queues are held in Amazon SQS and the arrays of worker instances are managed by RightScale. This allows the monitoring part of our service to detect when the queue gets too large and more instances need to be launched.

And Animoto's use of RightScale and the Amazon Cloud is not unique. As Ebay grew their user base, they transformed their internal scaling architecture at least three times to maintain the level of customer support they needed.

Likewise, Facebook was faced with a problem -- users, lots of them, doubling in number in 8 months from one million users in 2008 to over two million users in early 2009 (a problem that all providers of a service might like to have!) They had massive scaling issues preventing them from servicing their user base and hitting their service level agreements. Some simple changes, such as adding to their distributed data management caching using memcached, resulted in a 4x increase in services throughput, which removed their system's backend data management scaling constraint.

MySpace*, like Facebook, uses relational databases extensively front-ended by a layer of Memcached servers -- a nice scalable design with over 3000 front-end servers and well over 100 database servers (with one million users per database server).

Application Design Principles

What is going on here? They key message is that "Internet Scale" brings new software architecture principles. Many architects of internet-scale systems agree on a few key design principles:

- Design horizontally -- build software components that scale when adding new hardware platforms
- Federate data into fixed sized chunks (partitions, cells, sharding, etc.)
- For updating (writing) persistent data, design with eventual consistency in mind (rather than strong consistency as demand by ACID properties on global database usage) so there is not a single global
- Cache (especially for computed data close to its user) to limit the load on other entities (especially the persistent storage layer)
- Queue to disconnect functional work

How do these desired components of a software system's scalable design map back to real hardware capabilities? Because these key design principles are tailored to place a computation package (an "entity") in one platform and then scale that entity/platform to meet demand. Even though many details of the hardware are quite rightly shielded from the users, some items, like performance and capabilities are related to entity sizing.

Why there are limits are quite clear – if each individual compute node is overloaded (lets say it's in a state of continually page thrashing), very little forward progress will be made by the application, no matter how many compute nodes it is granted from the controlling Cloud operating system. This makes it quite clear why your application testing needs to ensure your application can configure itself to the hardware it runs on. Likewise, entities define to run on over-configured hardware specifications are going to cost the users more to charged-for resources to execute and are likely to limit scaling for all users.

In addition, the insistence of the design on middleware layers (caching, queuing, etc.) to mitigate load on the data persistency layer, raises an opportunity for a more heterogeneous platform landscape with many different properties (memory size needed, i/o bandwidth needed, etc.) than the applications' data or business logic layers. Again, as an application architect and designer, each component of this landscape needs to be properly sized.

The key point here is that performance and scaling needs should not be projected out to the ridiculous conclusion that "the hardware doesn't matter", but should be update to the objective of "how can I ensure I can configure my application into a set of logical entity types that comfortably fit in the currently available infrastructure nodes". We will call this "entity sizing" problem.

Note that this problem is only secondary to the problem of scaling or determining dynamically how many entities are needed to meet a given demand. The primary scaling problem is to dynamically update the number, say N, of entities of a single application type, say "foo" that are current running in the cloud to service demand load. The secondary "entity sizing" problem, rather, this is how to size each "foo" instance so it doesn't overload a "T-shirt sized instance" in the infrastructure.

How to Size and Configure: The Entity Sizing Problem

Let's examine some suggested actions you can take in using these principles in your architectural design and in your internal benchmarking efforts (performance, sizing, service scaling, ..) to get good packing fit of application components entities types to infrastructure capabilities.

Action1: Add ability of your application to have entities with configurable computation needs

This is the basic idea of applying the architecture principles. At the database level, you shard or partition your data in some logical way. At the business logic level, you define single OS thread pools which logically serve similar requests from multiple users. At the web-tier layer, you configure redundant grids of application servers that distribute load. At the caching level, you determine points where the application repeated access the same data from persistent storage or from commonly computed results.

Action 2: Add testing phase to your development using a workload to set entity configuration parameters and define the computational resources required to meet an entities' goals

This is the key to entity sizing. You need to determine fit of your application components to the underlying hardware resource capacities.

A major requirement to do this properly is a good workload or benchmark which can drive load to at least one instance of each application entity type. Workloads can be hard to manufacture, but having one, even if synthetic, will bring a source of data rather than guesswork to your configurations.

The most common idea is to then to setup some pre-defined hardware platforms targets of various capabilities, call "T-shirt sized" configurations, in other words, "small", "medium" and "large" configurations, which vary the amount of processing power, memory, and I/O capabilities available. These are generally "technology sweet spot" configurations, available from major Infrastructure as a Service vendors (such as Amazon Web Services) or from major OEM platform manufacturers. Given that this technology changes rapidly, its advantages to use the most recent hardware offerings to define your configurations.

Testing consists of determining the size of an entity in application specific configuration terms (for instance, amount of data stored, maximum number of users configured, maximum number of jobs queued, etc.) that maximal "fits" on each "t-shirt" size platform while still maintaining end-user SLA (latency, throughput) requirements.

Action3: Add capabilities in your entities to "configure itself"

In more advanced testing, you could determine platform configuration model parameters (instructions per transaction, ratio of memory per CPU thread needed, I/O bandwidth needed per transactions, etc) so that you have general model of platform size need for any load. For this mode, you may want to have a local set of hardware platforms that allow you to modify individual configuration parameters, rather than rely on "t-shirt" configurations from your cloud service provider.

Once this analytic model is available, entity configuration can be applied dynamically at entity provisioning time on a particular virtual platform.

Action4: Add horizontal "flood plain" benchmarking to your performance & scalability testing repertoire

Many companies size their whole applications as above, by driving load on a single platform till that platform breaks, and take that as a static measure of platform capabilities. This is generally called "high watermark" benchmarking.

Note that a high water mark refers to the highest level reached by a body of water that has been maintained for a sufficient period of time to leave evidence on the landscape. It's quite hard to see the high water mark if you standing in the middle of the flood plain -- in fact the concept has less meaning when you're looking down on water rained from the clouds and you can see with no obvious markers for the how high the water is.

For clouds, your high watermark benchmarking will need to evolve into "flood plain" benchmarking. By a "flood plain" benchmark, I simple mean a benchmark which demonstrates and measures performance attributes as the flow of computation responds to demand, i.e., as the computation "flows" to overflow platforms to meet increased demands. This is a key property of the advantage of cloud computing infrastructure.

The flood plain example becomes analogous to a key change in thinking of the purpose of the cloud computing application architecture infrastructure: The goal is no longer:

- Given a set of hardware resources and guaranteeing full application data consistency, minimize the response time of requests and maximize the throughput of requests.

But rather:

- Given performance requirements of an application (peak throughput; maximum tolerable response times), dynamically minimize the required hardware resources and maximize the data consistency

By adding flood plain benchmarking, you will need to look at new and additional performance items:

- Impact of parameters underlying the guiding architecture principles you used from above (size of caches, size of queues, servers per pool, etc.)
- Ramp up and down times for services,
- Impact of parameters (timing frequency, etc.) of monitoring & feedback control loops

Conclusion

Applications need to effectively utilize the capability of the compute nodes that make up their delivery cloud computing infrastructure. One way to achieve this is to focus on the concept of application entities. This concept has also proven effective in achieving "internet scale".

The size of these entities must be defined and configured to work within the constraints of capabilities of a single OS image running on real-world hardware. If they do not, they can no longer compute effectively.

While we have stayed at a very high level of action recommendations in this short note, we have suggested actions which impact application design and testing, actions which stress the importance of mapping application entities to platform capabilities, actions which are needed to effectively delivery application services into any cloud computing infrastructure.

About the Author

Dr. Stephen P. Smith is a computer scientist at Intel working with the ISV ecosystem to ensure industry software runs best on Intel hardware. In the past, he lead Intel's factory Planning and Scheduling automation group, and was a senior research at Northrop's Research & Technology Center. In his spare time, Dr. Smith reads science fiction and thrillers and pretends he can spike volleyballs.

References

[1] Animoto Scaleup example: http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up/
[2] Facebook scaling example: NYTimes: Is Facebook growing too fast
[3] Facebook: architecture changes to meet scaling: http://www.facebook.com/note.php?note_id=39391378919

[4[MySpace architecture:
http://perspectives.mvdirona.com/2008/12/27/MySpaceArchitectureAndNet.aspx
[5] eBay architectural history: http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf
[6] eBay scaling best practices: http://www.infoq.com/articles/ebay-scalability-best-practices
[7] Design principles to scale services: http://www.cs.berkeley.edu/~brewer/Giant.pdf
[8] design principles: Building Scalable Web Sites, Henderson, O'Reilly, 2009.
[9] design principles: Enterprise Web 2.0 Fundamentals, Sankar and Bouchard, Cisco Press. 2009.
[10] Design principles: Life Beyond Distributed Transactions: http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf

* Other names and brands may be claimed as the property of others