The Need to Comprehend Clouds: Why We Still Can't Use Clouds Properly

Daniel Rubio Bonilla[1], Lutz Schubert[2], and Stefan Wesner[3]

[1] HLRS, University of Stuttgart,
Nobelstr. 19, 70569 Stuttgart, Germany
rubio@hlrs.de
[2] IOMI, University of Ulm,
89069 Ulm, Germany
lutz.schubert@uni-ulm.de
[3] KIZ, University of Ulm,
89069 Ulm, Germany
stefan.wesner@uni-ulm.de

Abstract. Clouds have become the modern concept of utility computing – not only over the web, but in general. As such, they are the seeming solution for all kind of computing and storage problems, ranging from simple database servers to high performance computing. However, clouds have specific characteristics and hence design specifics which impact on the capability scope of the use cases. This paper shows which subset of computing cases actually meet the cloud paradigm and what is needed to move further applications into the cloud.

Keywords: Cloud, Use Cases, Cloud Dwarves, Cloud Performance Criteria.

1 Introduction

The cloud concept allows reacting to system load dynamically to distribute the services according to actual usage, thus reducing the cost of ownership and leading to better resource utilisation. Clouds have become the modern paradigm of utility computing. At the same time, with the rise of GPU computing and multicore processor architecture, there is a growing belief that performance is proportional to the number of resources. It is thus frequently assumed that clouds can implicitly increase the performance of applications.

This assumption is however wrong for two major reasons: (a) performance is not generally proportional to number of resources and (b) applications do not simply change their behaviour (and thus quality criteria), just by being deployed in the cloud. There has been an abundant discussion on scalability and performance limitations, which shall not be repeated here (see e.g. [1]). This paper will elaborate why these limitations apply and which effect they have on the usability of the cloud for different application scenarios (section 2). It will give an assessment of the difficulty and expected value of migrating use cases to the cloud and will provide a first approach for classifying them regarding their benefits from clouds (section 3). We will show in particular that many factors regarding the relationship between code and cloud behaviour are effectively still unknown and outline the necessary work that needs to be done in order to improve future exploitation of cloud systems (section 4).

2 Cloud Delimiting Factors

To really exploit the full potential of cloud environments, it is absolutely necessary to first understand what clouds are and thereby which capabilities they really offer. Even though the concepts are widely known, the principles behind their realisation, and thus their limitations are less well documented. This is due to the quick uptake on the market, as well as because there is no reference technology for realising clouds, though Amazon EC2 and Google Docs are the de facto reference infrastructures.

According to the cloud report published by the European Commission, the primary cloud characteristics are specifically [3]:

- Utility Computing
- Elasticity
- Availability & Reliability
- Ease of Use

2.1 Size and Interconnect

A cloud environment must thus consist of multiple computing systems that can dynamically host multiple instances of the same service / application. In other words, that can replicate the functionality offered according to the current demand, and also reduce it in a similar fashion. Typically, this is realised by exploiting virtualisation technologies that host the respective logic, but can be easily encapsulated and therefore moved between instances, respectively replicated as a full image. The main point is that this behaviour is transparent to the user (i.e. does not require reconfiguration of their systems) and that it is steered according to the load, respectively availability requirements.

Due to technical constraints, elasticity is considerably slow, as distribution of the image typically involves communication of a large quantity of data, in particular if the image packs the whole operating system and execution environment of the service in question. With increasing complexity of the service and its runtime environment, the according delay implicitly increases and thus makes reaction to availability requirements slower. This is however a technical constraint posed by the typical implementation approach, and not by the cloud concepts as such.

As more and more applications move to the cloud, more and more users access internet-based services and the scale of individual applications increases to compensate the performance needs, the technical constraints become the major impeding factors for the fulfilment of the main cloud characteristics. This means that the number of resources available in a cloud environment may quickly become insufficient for the needs of the services, respectively users – in particular at times of peak demand. This leads to the same resource utilization problem again that kicked off the cloud concepts in the first instance.

Communication limitations become serious impeding factors for performance of web-based services. Not only the degree of sharing between connections and users, but also distance between server and client play a major role for this factor. Whilst downloading a file within your own country may reach a bandwidth of multiple MBps and a latency of less than 10ms; for foreign countries, depending on distance and connectivity, this may decrease to kBps and latency of a few hundred milliseconds. For a Gigabyte file this can make the difference between minutes and hours.

Latencies of milliseconds sound comparatively little considering the acceptable delays of about 1 second for loading and displaying a website without interrupting the user flow [4]. However, this latency is a constant, adding to any transaction between server and client. Implicitly, whilst it has hardly any impact on large, it creates massive delays for any interaction that bases on multiple communication exchanges with comparatively short messages. In particular for real-time interactive applications, such as MMO games, this built up in latency leads to massive problems.

Table 1. Network performance in different environments

	Latency	Bandwidth
Internet	150 ms	10 Mbps
Server Farms	10 ms	1 Gbs
High Performance Computing	<1 ms	up to 100 Gbs

Notably, latency and bandwidth are subject to physical constraints (cables, speed of light, routing etc.) that start to catch up with the requirements. It is of interest to compare the situation for average internet access with what is currently possible on the high end of the scale. Table 1 summarises the typical network performance parameters in three typical domains – obviously the figures may differ according to the server's capabilities, its load etc.

What is more important than the concrete figures is however that these domains differ by 1-2 orders of magnitude in performance, i.e. internet is 10 times "slower" (to react) than intranet, which in turn is (more than) 10 times slower than high performance computing. Similarly for the bandwidth, where internet's capabilities are 1/100s of the intranet, which in turn is roughly 1/100s of High Performance Computing (HPC). We must in this context distinguish between the cloud resources (typically server farms), and connection with the end-user. So that cloud access to the cloud, and potentially between clouds is internet-based, whereas resources within the cloud can reach an interconnect of the speed of server farms.

Notably cloud systems try to reduce the communication limitations further by compensating for distance and concurrent usage through replication and relocation of the service instances. Even so, the inherent (physical) restrictions cannot be overcome, but are the core constraining factors. What is more, the level of freedom, i.e. the degree of influence a management system can take, is higher with "higher-level" domains, than with low-level ones. This is partially due to the fact that the performance is actually achieved by maximum alignment of the system layout with the application cases – factors such as uncontrolled concurrency, or even shared resources are thus not supported in these domains in the first instance.

2.2 Execution System / Middleware

In order to allow for the dynamic distribution and instantiation of services, respectively applications, it is necessary to be able to package them and re-instantiate them with their full context. Virtualisation allows not only to host the full service environment, but also pausing, replicating and moving it with little additional overhead and with little impact caused by the underlying hardware.

However, virtualisation technologies limit the performance of the actual hardware, restrict the scalability and cannot easily share data and / or code between instances. As such, the most straight-forward usage for virtualisation consists in providing completely isolated images, where every user accesses his own instance. In cases where applications share data (e.g. Wikipedia like social environments) or even parts of the logic (such as in stock market analysis), solutions become more complicated: In these cases, it is more advisable to exploit an execution framework of its own, dedicated to the respective use case and on which specific data and configurations, rather than code is enacted. This means that every user is effectively using the same logic with different distributions and instances of data and shared algorithms.

A similar approach can be used to expose a dedicated application programming interface for the respective usage domain that allows the user to develop his / her own logic on top of a (cloud) managed infrastructure. This allows best adjustment to the underlying infrastructure and management of the enactment according to the specific domain requirements, but it at the same time limits the application scope.

The essence in these approaches is similar: to completely retain control over the systems and in particular the execution of the hosted logic – only in this fashion is it possible to realise the essential cloud capability, namely the dynamic adaptation to load criteria. The elasticity focuses specifically on the number of instances to be replicated in order to fulfil the respective quality of service criteria. The management and adaptation framework must thereby be well adjusted to the actual application case, in order to enact the required consistency mechanisms for shared data, to reroute messaging according to the instance relationships etc.

Management and adaptation create additional overhead that reduces execution performance, thus restricting dynamicity considerably. Most cloud environments take therefore generally a pro-active and cautious approach towards elasticity, i.e. create instances ahead of time (i.e. before the availability criteria is threatened to be violated) and keep instances alive even after need, to reduce re-instantiation time.

Again we can make a comparison between different means of instantiating and relocating an application / service / image, though comparing mechanisms rather than domains (see Table 2). These figures thereby completely neglect additional overhead for communicating the associated data over the network as described in the preceding section. Essentially, with the complexity of the mechanism (e.g. virtual machines over processes) the amount of data that needs to be shifted with the new instance increases, too. The effective speed in the according domain is therefore reduced by the factor produced by the typical interconnect setup (see above).

Table 2. Instantiation / replication handling performance

	Delay
Virtual Machines	Minutes
Managed Processes / Services(PaaS)	Seconds
Threads (OS)	Milliseconds

Similar to the interconnect performance, we can note that there are multiple orders of magnitude difference between the individual mechanisms, which impact on the instance handling speed accordingly.

2.3 Programmability

Programmability is a major issue for full usage of cloud systems. Keeping the wide customer base in mind, the programming language should adhere to well-established models, such as Java, C#, PHP etc. Notably, the language of the application itself is secondary, if full-fledged virtual images are provided (IaaS), but is of major concern in PaaS environments, where the extensions typically adhere to a specific language.

Stand-alone, non-adapted code versions work fine in IaaS cases where each instance can be treated completely isolated. Once dependencies, i.e. shared data is introduced, the application logic needs to be altered accordingly. If the developer wants to control specific features of the cloud behaviour (such as scaling behaviour), the according knowledge needs to be encoded right into the logic. It is worth noting in this context that not all cloud providers automate the elastic behaviour, but expose an according programming interface to the developer to trigger instantiation himself.

The main task therefore consists in rewriting the logic to externalise content and part of the logic. One subsequently introduces delays for data-exchange, which is proportional to the factors discussed above. A straight-forward approach may consist in externalising the database system and sharing it between instances, but this will create a bottleneck, which may counter all benefits from migrating to the cloud in the first instance. Introducing dedicated synchronisation points similarly leads to communication delays that delimit the execution performance.

Modern programming models allow for easy integration of web interactions and cloud features, but the relationship to the actual performance and behaviour is not clear. In other words, the available languages are not able to compensate for the deficiencies introduced by the algorithm itself. For example it makes a major difference whether the developer intends to share a large database that is updated multiple times, or whether the application effectively just exchanges data at session begin or end time. The language allows for either way without giving indicators of the performance impact, let alone controlling this impact. Most languages completely rely on the framework, respectively middleware to execute the transactions. This means that the user cannot estimate the timing impact correctly.

Lower level domains, such as HPC, therefore do not rely on managed communication frameworks, but essentially leave full control with the developer. Extensions provided through MPI or OpenMP only expose standardised mechanisms for common procedures, rather than taking responsibility away from the programmer. Thus, the program has to be instance aware to deal with the dynamic instantiation and relocation – typically it is therefore only exploited under very controlled conditions.

To optimally exploit the cloud characteristics, partial and conditional sharing of both code and data would have to be supported and ideally widely automated. No current programming model allows for such support though and the manual approach currently undertaken by e.g. HPC is highly complex, leaving only very few people ready to deal with it in the first instance.

2.4 Consequences

Considering the primary characteristics of clouds, it is obvious that full exploitation of the capabilities is tied to the use cases, even though the principle allows for a broader scope with the implicit loss of performance, respectively quality of service. Clouds are often treated as the panacea of IT, but they are effectively specialised domains, so that outside of this domain other environments show better performance.

These domains, however, cannot be easily specified along the line of "eScience", "business applications" or similar. Rather, they deal with specific immediate requirements (availability, elasticity etc.), thus allowing for a wide scope of use cases all following under "utility computing". But similarly, "eScience" or "business applications" cannot easily be translated into one specific IT domain either.

We can note, which conditions the application should fulfil in order to be able to exploit the cloud, respectively to benefit from it in the first instance – these are:

- Dynamic number of users, respectively requests
- Small and infrequently shared data between a limited number of instances
- Communication between instances and data sharing is primarily asynchronous
- Comparatively low scale of the application itself (i.e. degree of parallelism)

As noted, many of these conditions can be seen as comparatively "lax" boundaries, i.e. they allow for a certain degree of freedom – e.g. synchronous messages are certainly possible, if the delay is not crucial for the execution of the application. We can however also denote hard boundaries that cloud systems cannot fulfil and therefore constrain the scope of usage:

- Single instance applications simply do not benefit from cloud management
- Data intensive applications where performance is crucial
- Large scale applications which require a large amount of resources for fulfilling their work. Notably, they might run on the cloud, depending on their communication dependency, but they do not exploit the essential cloud capabilities
- Applications where execution performance is crucial and where performance is influenced by any communication related overhead, including instantiation

3 Classifying Your Application

As elaborated in the preceding chapter, clouds are constrained in their applicability scope and many use cases either do not benefit from the additional capabilities offered by the cloud, or even suffer from its limitations. It has also become obvious though, that it is not easy to classify an existing application or use case for its potential benefits from cloud environments. Most of the core characteristics identified in the preceding section may be hidden within the application, i.e. it requires real in-depth expertise of the application to identify it. What is more, it is not clear whether the according constraints could not be overcome by changing the code (see next section).

Here we provide a set of criteria that may help in classifying the use case / application and to assess the potential benefit to be gained from the cloud:

3.1 Classification Criteria

With respect to the preceding assessments, three major criteria stick out:

1. Scope (degree of sequentiality, respectively parallelism)
2. Strength (or "tightness" of the communication between instances)
3. Density (or amount and size of exchange / communication between instances)

These three criteria can more or less be directly mapped to instantiation speed, communication latency and bandwidth as elaborated in the first section:

Scope. The number of instances required and the number of instances maximally possible (i.e. scalability) define the resource "hunger" of an application and therefore its need for a larger infrastructure at all. We thereby need to distinguish between the resource need of a single instance (parallelism) and the amount of instances required due to the amount of requests / users (concurrency). The effective need is therefore the product of parallel scope time concurrency scope.

Strength. The acceptable communication delay for any shared data access or information exchange provides an indicator for the acceptable latency and therefore for the type of infrastructure required. We can most of all distinguish thereby whether the exchange is synchronous and therefore directly impacting on execution performance, or whether the communication can be executed asynchronously, in which case the impact on performance is considerably less.

Density. The communication delay in itself may have little impact if the amount of messages, respectively data accesses is comparatively low and if the messages in themselves are comparatively small. For example, even a synchronous, blocking request of multiple seconds duration may be ignored, if it only occurs a few times per hour, so that delay << execution time.

3.2 Analysing the Use Case

The benefit of the criteria provided above is that they can be extracted from a given application in a fairly easy way, though their interpretation may not be as straight-forward as the conditional cloud characteristics listed in the preceding section. In the following we assume that the application has not yet been migrated to the cloud, though the same principles would apply:

As has been noted multiple times, the cloud is an infrastructure consisting of multiple dynamically allocated servers that can host an elastic number of application and data instances, according to requirements. In order to exploit this infrastructure, the following migration scenarios are possible:

1. Keep the application standalone, sharing no data or code
2. Share data between instances to allow for collaboration, networking etc.
3. Distribute the code to allow partial sharing of functional logic
4. Distribute code and data over the infrastructure

The general idea is to make use of multiple resources thereby creating a better user experience. The simplest case is obviously 1), where no further actions have to be taken and each instance is simply hosted in its full environment (image) - this provides the service with considerable enhancements of availability through the cloud infrastructure, but provides no further benefits. For example, this does not allow users to share data with each other, nor does it allow to make efficient reuse of intermediary processing results, such as in live rendering - in short it does not allow for any collaborative enhancements. It also does not allow for exploitation of concurrency between individual functions, so that generally this creates no performance benefit.

Most enterprises and individuals consider transition to the cloud to improve service quality however and want particularly to improve execution performance, reduce resource costs to their minimum, and offer collaborative features. In such a case, any of the options 2.-4. may apply, whereas complexity increases with the higher options.

To identify which options are possible, it is necessary to analyse the use case in question. To this end, the dependencies between functional and data units need to be analysed with respect to their potential for being distributed. Whilst model driven architectures do provide some insight into these dependencies, the actual impact on the criteria listed above can only be estimated and may vary during execution.

To gather meaningful data, the best approach consists in instrumenting the code or by monitoring the memory behaviour as recommended and elaborated by the S(o)OS project. The S(o)OS project furthermore indicates how this information can be interpreted to the purpose of code segmentation, distribution and parallelisation: by weighing access frequency and read / write patterns the degree of dependencies (strength and density) can be qualified. Using this weight, segmentation cuts may be performed in the code according to the communication (and exeuction) capabilities of the destination platform [5].

We need to extend this model to investigate the impact of more or less arbitrary access attempts via the externalised interface. This behaviour can either be simulated and evaluated through actual memory usage, or propagated along the weighted graph in the form of heuristic representations of the user behaviour.

The analysis of the respective application should in all cases be able to indicate access frequency and size (density), as well as the dependency between request and actual usage (strength). By furthermore annotating the data with respect to whether it is expected to be shared between users, the analysis thus provides a fairly accurate distribution architecture, including communication requirements and instance control indicators. The according mechanisms will be investigated in the upcoming EC funded research project PaaSage, and published shortly.

3.3 Interpreting the Results

Basing on the analysis, the use case's "cloud potential" can be assessed (respectively the most suitable platform can be identified):

The code can principally be segmented where the expected control behaviour changes substantially (i.e. a boundary between shared and non-shared functions / data). For example, the graphical user interface of an application can typically be easily segmented from the actual logic. In the application analysis graph this is denoted through individual memory spaces per each external instantiation request, rather than a shared memory region. These separate regions are an indicator that either code and data, or just data can be maintained per individual instance, whereby the trigger in this case may be the external request, i.e. user connection.

The impact from segmentation and clustering can be assessed by measuring / calculating the implicit changes in strength and density of the connection and countering it with the scope of the instance, and therefore resource need. There is accordingly no strict boundary between use cases that are suitable and those that are not suitable for cloud environments. Just like in any IT case, execution can be enforced even in "foreign" environments, but at the cost of performance. Accordingly, the boundary is implicitly given by when the loss (performance, migration cost etc.) outweighs the gain (elasticity, availability etc.)

The table below may serve as an indicator for which freedom in the terms of the three main criteria can be considered acceptable for a cloud environment, before the loss will start to outweigh any potential benefit. Note that table 3 is not providing clear boundaries either - instead with lower rows in the table, the likelihood of benefitting from cloud environments diminishes.

Table 3. Suitable infrastructures for specific criteria combinations. Note that the "environment" transitions smoothly from Clouds via Server Farms / Grids to High Performance Computing.

Scope	Strength	Density	Suitable Environment
small scale, many instances	low	low	Clouds
small scale, many instances	medium / streaming	low	...
medium scale, few instances	low	medium	...
few instances, medium scale	medium	medium	...
...			...
large scale, few instanced	high	High	HPC

There is a high risk with this classification to neglect data dependent behaviour, i.e. the same code may exhibit different strength, density and scope, depending on the data it is processing. For example, the input parameter may directly specify the number of instances to create. If this relationship is not known, the impact of this factor should be evaluated through analysis of common data structures.

4 Making Your Application Cloud-Efficient

Whilst the analysis provides some insight into the potential of a use case to be migrated to the cloud, it does not help to assess alternatives, i.e. whether the code could principally be restructured to exploit the cloud features more effectively. For example a mathematical algorithm with high data dependencies will create a high requirement for density and will be fully impacted by the effective communication strength. But a large iteration over individual (i.e. data-independent) actions, such as extensive independent calculations on a parameter range, may be easily distributed, as the requirements for communication are comparatively low.

By choosing another implementation approach, the same application may expose characteristics befit better the cloud. We can consider this a functional transformation from logic A to B in a way that B exhibits a different set of criteria, more suitable for the specific destination platform. Transformations such as this are very typical for parallelisation efforts, where the developer attempts to make his code suitable for HPC clusters in order to exploit the additional performance of supercomputers.

As has been shown, the key driver for cloud based applications is not performance, but concurrency, availability and elasticity. The terminology of cloud providers is often highly confusing with this respect: when talking about scalability, performance and availability, they generally refer to the level of services typical for the web domain. A user may however easily expect scalability and performance to the degree of HPC. It must be stressed though that within the web service domain, clouds offer a major improvement over alternative approaches. To effectively migrate applications to the cloud, it is of utmost importance to be aware of this difference and to build the application around these primary constraints.

On this basis, core functional building blocks can be identified that are most suited for the cloud environment, respectively that relate better to other domains. Conversion means can be identified that help identification and conversion of specific routines to meet the environment's conditions best. Such information would allow the developer to choose the right algorithms at design time; at the same time it would provide indicators as to whether a certain application is principally suited for the designated environment. Such efforts have been undertaken in the domain of High Performance Computing for considerable time now, such as Berkley’s report on key algorithmic cores [2]. A similar attempt for clouds is now being initiated by the EC funded project PaaSage, but essentially the necessary expertise is still lacking as of today.

Some indicators can, however, already be identified and may serve as a basis for further elaboration. These indicators obviously relate strongly to the key criteria identified within this paper, namely: scope, strength and density:

Scope (and implicitly scale) is impacted by the degree of sharing within the application. One way of improving scope therefore consists in reducing strength and density, as discussed below. Another way of improving scope consists in clustering the logic, in alignment with the data dependencies, thus generating a modular software structure that shows different scale within the different modules.

Strength is a major hemming factor in efficient usage of clouds, as the communication delays must implicitly degrade performance. Programming models generally do not allow differentiating between time for fetching and using data, thus leading to the assumption that effectively data is available immediately. The effort to compensate for strength must instead be taken by the developer.

The best way to approach this consists in using asynchronous messaging right from design time. This automatically forces the developer to organise the code in a fashion that caters for weak strength communication, and to exploit idle time for performing other tasks. Asynchronicity also means that eventual consistency should be considered for shared data, rather than immediate consistency. Notably, switching to asynchronicity is not always possible, which could mean that the delay is secondary, or that the application is simply not suited for cloud infrastructures.

Density plays a particular role in combination with strength: multiple messages or huge messages can sum up in the performance degradation. Even if coupling is weak, the mass of communication will create an impact. The key point for the developer is to identify the right mix of number and size of messages – e.g. by packaging multiple small messages into one bigger to reduce the impact of latency, or by splitting up bigger messages to deal with bandwidth limitations.

Developers should carefully evaluate the impact of multitenant behaviour upon the application: generally not all data needs to be fully shared, but only within groups of instances and here only parts of the data at different times. Keeping an eye on the specific intention is a good way to reduce density concerns. For example, Google docs share documents between groups of people, yet the only data that needs to be communicated is the changes that are actually taking place in the document.

5 Conclusions

Clouds have been around for a considerable time now, but there still exists little knowledge about their actual capabilities and limitations, let alone about how to address them and which use cases are most suitable for it. More expertise needs to be gathered about the essential core application logics that are most suited for clouds. These core elements can be used as a basis to build up cloud applications, but also as a means to quantify existing applications to assess their "cloud-suitability". The paper presented provides an initial outline for identifying these cores by classifying the main criteria constituting cloud application performance and behaviour.

Acknowledgments. The work presented in this paper was partially funded by the EC research projects S(o)OS and PaaSage.

References

1. Huonder, F.: Parallelization for Multi-Core. Program Analysis and Transformation (2009)
2. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. EECS Department. University of California, Berkeley (2006)
3. Schubert, L., Jeffery, K.: Advances in Clouds - Report from the second Cloud Computing Expert Meeting. European Commission, Brussels (2012)
4. Miller, R.: Response time in man-computer conversational transactions. In: Proceedings AFIPS Fall Joint Computer Conference, vol. 33, pp. 267–277 (1968)
5. Schubert, L., Kipp, A.: Principles of Service Oriented Operating Systems. In: Vicat-Blanc Primet, P., Kudoh, T., Mambretti, J. (eds.) GridNets 2008. LNICST, vol. 2, pp. 56–69. Springer, Heidelberg (2009)