Framework and patterns facilitating cloud computing adoption

Ciprian Crăciun
Scientific adviser: Prof. Dr. Viorel Negru
West University of Timişoara, Romania

1 October 2010

Contents

Chapter 1

Introduction

Today one of the *"noisiest"* topics that has captured the attention and managed to actively involve both the industry and academia is *"cloud computing"*, a name (*"cloud"*) well suited in relation with the fuzzy-ness of the subject, as the domain is still in its infancy, and is barely starting to take shape.

Also interesting about the topic are the reasons behind each actor's involvement: the industry seeing this as a potential solution for both technological-enhancement as well as cost-efficiency, meanwhile the academia wanting to further apply and develop branches of many involved disciplines (like networking, operating systems, distributed computing, artificial intelligence, etc.) and building on previous experiences in clustered and grid computing to revitalize and continue these respective topics.

Another achievement — for some maybe outweighing all the others — is the commoditization of software and hardware infrastructure, a breakthrough similar to the commoditization of personal computers, because it enables anyone with a great idea and a small budget to build and actually deploy it on a performant (though virtual) infrastructure without needing the initial investment. This is the case of most start-ups that were early adopters of this solution, and based on their internal flexibility and agility soon ran shoulder-to-shoulder with the big companies, and eventually capitalizing their success by either growing or buy-outs. Thus we might consider these small start-ups as a third actor in the cloud computing ecosystem.

All these factors contribute to the *"noise"*, each actor being interested in what cloud computing offers particularly to him, each pushing or pulling the topic in their own direction. As a consequence the topic is maybe the most dynamic, full of surprises and turn-arounds than any other current research domain — just like the invention of the Internet and the fairly recent `SOA` movement.

Thus I would personally say (and hope) that the environment is very fertile for research and development, and one that will shape the future of computing with many impacts on industry and humans in general — not as a big leap as the Internet, but maybe closer. Also even though the subject is technological in nature, it leaves room for ideas and developments of many types: philosophical (by changing the way we think, build, and use computing), theoretical (by finding solutions to problems that arise either from direct needs and challenges or from more long term vision) and engineering (by melting and merging all these ideas into pseudo-tangible, as they are virtual, systems that we can interact with).

Therefore confident in my choice, I've put time into better understanding this domain, starting from a *"10.000 miles"* grand-picture of the whole thing, and then closing-up on those subjects I find interesting — knowledge that summarized in the second chapter — thus bootstrapping my research activity by having identified those topics that I shall invest energy into — summarized in the last chapter — hopping that my outcomes will contribute to the state of art.

Chapter 2

Cloud computing

2.1 Overview

Defining cloud computing (at least at the current moment) is a very difficult task, for at least two reasons. First because it is a very hot research topic which has just started to take shape, and it will take some time until the industry and the academia is able to explore all the directions and crossroads with other fields. And second because it caught the general industry's attention as a potential revenue driver, both as a technological leverage — allowing them to optimize or cost-efficientize their internal operations or product offerings — but also as a marketing *"buzz-word"* — by stamping the *"cloud-enabled"* label on anything from mere load-balancers, clustered application containers, replicated databases, to virtualization technologies.

But before citing any definition it would be better to start off by citing a warning [15]:

> Cloud computing is still an evolving paradigm. Its definitions, use cases, underlying technologies, issues, risks, and benefits will be refined in a spirited debate by the public and private sectors. These definitions, attributes, and characteristics will evolve and change over time.

Thus if searching for such a canonical or definitive definition may be impossible, we can at least settle with one generally accepted (both by its broad coverage, and by its proponent position and acceptance) — the definition given by `NIST` which states [15]:

> Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Which is then followed by the essential characteristics that summarize the definition:

> [...] on-demand self-service, broad network access, resource pooling, rapid elasticity, measured service [...]

Another recent definition — that shares the spirit of the previous one, but which clearly points out the scalability requirement, business model, and resource contracts — is provided in the paper [17], together with at least a dozen citations of earlier definitions (the before mentioned paper also represents a broad overview of the subject):

> Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and / or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an

> optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs.

Now that we have seen *"what cloud computing is"*, it might prove useful a definition of *"what cloud computing is not"* [**?**] — which I've chosen to cite because it captures a very important aspect of cloud computing *"automatic scalability"*:

> If the system requires human management to allocate processes to resources, it is not a cloud: it is simply a data centre.

Unfortunately if we search deeper into the subject of defining what cloud computing stands for we would find even more diverging definitions and reaching a conclusion would be quite difficult, as for example some proponents see it as an embodiment of utility computing, while others see it as rather a version of the old-and-tried grid computing [17]. As a consequence the best (and most accurate?) definition we could provide for it — and in general as to all things in real world — would be to analyze the way we can / should use *"the cloud"*, and then try to conclude.

Therefore in the remainder of the current chapter I'll try to focus my attention to the following subjects (each having its own dedicated section):

- *scenarios* — shall be a succinct analysis of how we are supposed to (and often do not) use the cloud, what are the most suited applications, and what are the advantages or disadvantages of (properly) using the cloud;

- *technologies and challenges* — in which I shall take one by one the core components that lay at the foundation of *"the cloud"*, for each of which I shall try to describe their fundamental architecture, weaknesses, strengths, proper (and improper) uses;

- *cloud mindset* — which should conclude upon the previous two sections, by describing the way cloud applications should be designed and implemented in order to take fully advantage of what is achievable; thus forming a complete image of what cloud computing stands for;

2.2 Scenarios

On the subject of cloud computing scenarios there is an abundance of technical reports, white papers, product data-sheets, guidelines, success stories, FAQ, etc. — as examples of comprehensive ones we could cite [2], [3], and [13]. The majority of them (if not almost all?) characterize the cloud from two main points of view: the *cloud access model*, and the *cloud business model.*

Before diving into those views, we must first define the actors involved in the *cloud computing economy* [17]:

- *cloud provider* — the entity — usually an commercial company, or to a less extent an academic institution — that owns a large set of physical resources / infrastructure, with no immediate planed use, and thus offers them for lend to other interested parties, as cloud resources (usually virtualized on top of the physical ones); they provide transparent (and in the near standards based) access to these resources, facilitated through easy to use web user interfaces, or through web service `API`'s;

- *cloud client / user* — again embodied by either members of industry or academia (they could even be interested individuals with the appropriate budgets) that register within the provider's systems without needing prior contact or relation with it (just a commonly accepted payment system); as a consequence for the client / user the *"getting started"* overhead is quite small; throughout this work these clients / users are going to be called either *"developer"*, or *"administrator"*, depending on the role which it plays in a certain situation;

- *final customer / user* — although not defined in other works — and based on the fact that cloud computing addresses mostly the business world than the academic one — the end user is the actual beneficiary of the entire cloud, as the businesses build, adapt and optimize their applications to satisfy the needs of their customers; (granted the exception that sometimes the *"final user"* is actually the *"cloud client"* itself;) thus both the cloud provider and the cloud client must take special care that the final customer doesn't need to know / care that the service he's using is powered by a cloud or is hosted in a dedicated data-center;

Coming back to the two aspects often pushed about cloud computing, we'll first tackle (the most important) operational benefits (and their respective technological implementations) [17]:

- *cost efficiency enabled by on-demand scaling* — by moving (both software and hardware) infrastructure *"in-the-cloud"*, a client is able to easily scale up and down as needed based on the current perceived usage; that is provision new virtualized resources (computing nodes, storage, or networking bandwidth) when the usage metrics (requests per second, consumed bandwidth, processor load, storage demand, etc.) show an increase (which is usually a spike), and of course decommission them when the metrics drop; this dynamism coupled with a *"pay-as-you-go"* model, allows a client to save both operational costs (by not paying for maintaining the infrastructure that it doesn't use) and capital expenditures (by not buying any initial infrastructure to start with);

- *simplified life-cycle management* — as most cloud providers offer to customers *"ready-to-go"* templates of many operating systems preinstalled with popular applications, the client (now in playing the *"administrator"* role) only needs to select such a combination (and if not available creating one is near to trivial); then once a template is instantiated and the resulting instance is deployed, the client doesn't need to concern itself with any additional configuration routines (network setup, security enforcements, regular updates, etc.) as they are either provided as part of the infrastructure or readily preconfigured in the templates; and non-the-least the client has at his disposal nice dashboards where he can monitor and control the virtual infrastructure with the press of a button;

- *increased availability* — because the instances are hosted by the provider on it's own infrastructure it will make it's best effort (as stated and enforced by the `SLA` (*Service Level Agreement*)) to keep the resources running at optimal parameters; and all this is usually transparent to the client and is once again achieved by leveraging virtualization (live migration), or applying well-known techniques like hot-standby, fail-over, replication, etc.;

Meanwhile other publications focus on another aspect of cloud computing — namely the way the clients interact with the cloud (the so called *"deployment models"* or *"offerings"*) [17] [2]:

- *`IaS` (Infrastructure as a Service)* — this is the most accessible and easy to use cloud format / technology, as all the client has to do is take his application, transform it into one (or a set of) cooperating virtual appliances, and then deploy it *"in-the-cloud"*; once more the key technology here is virtualization at the computing node (or sometimes the operating system) level: the client has complete access to his instance of the virtual node; he can install whatever application he needs, communicate with any other node (usually even with nodes outside of the *"cloud"*); thus what in fact most white-papers and guidelines are describing are instances of this scenario; needless to say even though this is the most flexible and easy to start up scenario, it does have its drawbacks, as the client (now playing the role of the *"developer"*) has to take into account and handle all the intricacies of having an entire node to manage: he has to take into account code upgrades, application failures, scalability issues, and stemming from those application start-up / tear-down procedures, load-balancing issues, etc.; thus the only way in which this differs from a dedicated data-center is the fact that the client doesn't need to buy and manage the infrastructure; (to summarize the client can play both the roles of an *"administrator"* and a *"developer"*;)

- *`PaS` (Platform as a Service)* — on the next level of abstraction we are offered a managed development and deployment framework, in which the client (now restricted only to the *"developer"* role) is shielded from messing with the actual (virtualized) hardware, operating system, or software frameworks; the cloud provider offers to the developer a customized `SDK` (*Software Development Kit*) composed of tools and libraries that help the him develop and deploy its application under this environment; the only worry the developer has about scalability is providing a budget for his application; thus by relinquishing the freedom of controlling the before mentioned low level details, he gains the freedom to focus on his business case and not caring about the virtualized infrastructure – he now uses a virtualized framework; (to conclude now the client only plays the role of the *"developer"*;)

- *`SaS` (Software as a Service)* — even higher on the levels of abstraction, the client now ends-up with a *"turn-key"* solution which he can immediately use; (he can of course also bother to customize the application's parameters to suit its own needs;) (in this case the client is reduced to a mere *"end-user"* with no / limited access to the actual framework or infrastructure;)

Now by analyzing these promises and models we can conclude — and many do so — that cloud computing is (only?) about virtualizing (nodes and storage) and moving them off-site in the hands of such cloud providers (thus keeping almost a 1:1 mapping between they old architecture and the newly virtualized one). Thus we are made to believe that the main (and sole?) use case for cloud computing is resource virtualization, but unfortunately this is only the most prominent part of what cloud computing can offer us.

Also the previous mentioned views miss a very important aspect of the whole domain: they present how we solve problems (the *"deployment models"*), how we account them in our budgets (the *"business model"*), but not what problems we are expected to solve (or not) (the *"actual problems"*) (they do provide generic statements, but none goes in depth with such an analysis).

As such we shall at least try to identify some form of classification for such applications based on the following orthogonal axes (in no particular order):

- *responsiveness* — how critical is the time frame in which an *"input"* once entered the *"application"* must produce an *"output"* (either in the form of a response, or some sort of tangible effect);
- *predictability* — if the resource requirements of the system can be predicted or even approximated (at least on short- to mid-term);
- *resource types* — what actual resources does our application going to use; what particular requirements or constraints we have from them;

On the *"responsiveness"* axis we have:

- *on-line (24/7) applications* — those kind of applications which once started are never stopped; they must *"run"* against all plausibly non-fatal obstacles (like server breakages, sporadic application bugs, customer originating load, etc.); as exponents of this category we have Web 2.0 / RIA applications which are facing requests from users, and which require low / medium latency with no (or rare) breakages, or they would drive users away; [8];
- *batch applications* — this category needs almost no introduction as we can fit in here almost all applications suited for parallel computing and (some) for grid computing; in general it's about applications that need to crunch high volumes of information (either as large data sets that need to be processed, or large problem spaces to explore), and which require no immediate results (i.e. the expected result turn-around-time is not in minutes, but in hours or even days);

Related to the *"responsiveness"* we have then *"predictability"* (as usually the on-line applications imply also unpredictable loads):

- *unpredictable loads* — where the application designer or administrator can't predict the resource needs, as these are needs are generated by external factors (e.g. human behaviour, Slashdot effect); [8]
- *predictable / constant loads* — the opposite of the above, where we can provision in advance; (but even in this case there are no 100% guarantees that the load is going to be constant, as enterprises never stop to grow, and scientific data never stops being produced;)

And on the last axis we have *"resource types"* (and these categories are not exclusive or disjoint):

- *computing* — applications needing computational power (again in this category fall a lot of classic examples from parallel / grid computing);

- *raw storage* — some other applications need to store large amounts of unprocessed data (like document contents, images, video streams, etc.); usually the requirement here is for such a store that is optimized for storing and retrieving a small number of large files, maybe accessing them in a sequential manner; (certainly random access falls out of this category;)

- *structured storage* — even though we could have merged this and the previous item into only one *"storage"* item, but they are kept distinct as the requirements and constraints of the two are irreconcilable; thus in this particular case the need of an application is expressed as frequent and granular retrieve and overwrite access to a very large set of values, identified by some kind of unique keys, usually following a random access pattern;

- *synchronous communication* — applications that during their execution need to cooperate by exchanging data in real-time (one component can't continue to work until it has feedback from another component);

- *asynchronous communication* — again, somehow similar with the previous category, but in this case a component can proceed (at least for a while) with it's job by just sending and enqueueing data, but not needing any immediate feedback before it can continue;

After having so categorized the possible cloud applications, and by going back to the previously described points of view, almost all of the applications described therein fall either under the on-line applications with unpredictable loads, needing moderate storage, and synchronous communication; either under the batch applications, needing computing and / or storage, with no / asynchronous communication. But unfortunately we must note that few (none?) of them speak about any communication / collaboration patterns between different components of the same application, as they treat applications not as a modular systems, but as monolithic cores that just *"sit"* above *"the cloud"*.

Thus to summarize things up — by taking into account that cloud computing is more focused to the enterprise world than to the academia — many of these information sources picture cloud computing almost completely from the business upper management point of view (`CEO`, `CIO/CTO`), and few pay attention to what actually means to *"build cloud applications"*.

2.3 Technologies and challenges

Thus in what follows I'll try to depart from the previous perspective, and move toward a more technical angle — closer to the researcher or developer. And by doing so I will describe one by one the most prominent (or ground breaking) methodologies or components that allow cloud applications to actually deliver the marketing promises (scalability, availability, ease of management, cost reduction, etc.) Also in this process I will try to take special care of two aspects of each item:

- *the challenge* — what is characteristic that makes the discussed subject interesting or non trivial to be solved; why can't the result be accomplished by other existing means;
- *the mindshift* — and this one is tricky to describe, in that I want to identify what needs the developer to do in order to take benefit of the discussed subject; what are the constraints or hidden pitfalls;

Before starting to describe the technologies I want to digress and explain why I consider this *mindshift* extremely important. In order to do this I will give two (related) well-known examples from recent IT&C history. (Unfortunately I don't have any scientific sources to back this up, but only the background noise raising from many corners of the Internet.)

- *`OOP` (object oriented programming)* — when it was first introduced it promised what earlier methods didn't deliver; one of these promises (and its main selling point) was code reusability: once a company would have invested in the design and implementation of a module / component (an assembly of objects and interfaces, with complicated inheritance and overriding relations), it could then have been reused throughout the entire company for all projects needing its services; unfortunately this was only half (some say even least?) achieved, and actual component reusability didn't raise to the expectations; now I would assert that (in many cases) the main barrier in achieving this were the actual designers and implementers — formed in the world of structured programming — who didn't just embrace the mindset of the object oriented programming, but instead they just kept their old habits and used the OOP languages as mere syntactic sugar;
- *`WS` (Web Services)* — (as in `WSDL/SOAP`) a few (technological) generations later (after `CORBA`, `DCOM`, etc.) there came the `WS` era which again promised what `OOP` had previously (and `CORBA`, `DCOM`, etc.): reusability plus interoperation; and again unfortunately this promise has not been fully achieved because of the same old reason: the majority of developers used `WS` as yet-another `RPC` (*Remote Procedure Call*) technology; and as a result today the majority of the industry has migrated to a different kind of `WS`: `REST-full` (*Representational State Transfer*) web-services, that try to diminish the `RPC` style, or to an asynchronous message-passing style like `AMQP` (*Advanced Message Queue Protocol*), proposed by the big players in the financial world;

Thus I strongly consider that in order to fully take advantage of what cloud computing can offer, we must not use it as a mere replacement for our old tools, but instead we must understand the intricacies, constraints, limits and pitfalls introduced by this new technology.

In what follows I'll try to cover the following topics:

- *distributed file systems* — giving enterprises the possibility to store and retrieve (with low / medium latency) exabytes of data;
- *distributed databases* — usually built upon the distributed file systems, offering a more abstract and high-level method of accessing data;

- *map-reduce* — the workhorse of cloud computing, based again on the previous mentioned items, standing behind Google's web search, and allowing them to churn all the Internet's data;
- *asynchronous communication* — although not very often mentioned as the previous two exponents of cloud computing, this communication style is also a key factor of the success of any cloud application;

However what I will not cover — mainly because I lack the knowledge about, or because my interest doesn't go into that direction, or merely because they are very well-known topics — includes: virtualization, `SLA` (*Service Level Agreement*), cloud economy, physical provisioning, load-balancing, and security.

2.3.1 Distributed file systems

When speaking of distributed file systems, we are mainly referring to the following (disjoint) categories (the last being the one we are actually interested in):

- *unified namespace networked file systems* — based on the classical client-server architecture, within which a file is hosted on a single or a group of mirrored servers, but which depart from the plain network file systems (like `NFS` (*Network File System*), `CIFS` (*Common Internet File System*), etc.) in that they manage to fit all the servers under a common namespace umbrella, so that the user has a transparent view of where the files actually live; exponents of these are the `AFS` (*Andrew File System*), `CODA`; thus we could say that in this case only the namespace is distributed, and not the actual content;
- *parallel / clustered file systems* — deployed in small-to-medium clusters ($\ll 20$ nodes, see in brackets the theoretical limits) comprised of high-end with low rate-of-failure hardware; there is usually no distinction between servers and clients; the nodes usually needing direct access to the block devices (generally hosted by `SAN` (*Storage Area Network*)); exponents of this category are `GFS` (*Global File System*) (≤ 8 EB), `OCFS` (*Oracle Clustered File System*) (≤ 256 nodes, ≤ 4 PB), Lustre, etc.; they exhibit high performance which is achieved only if all components are in optimal shape;
- *large scale distributed file systems* — and this is the category we are going to focus in the current section;

The first file system in this category was introduced by Google: `GFS` (*Google File System*) [10], and once the idea caught on, "clones" started to appear: `HDFS` (*Hadoop Distributed File System*) (part of Yahoo's Hadoop framework, and open-sourced as an Apache project), Cloudstore. Another non-`GFS` based is Ceph [20] (actually a clone of Lustre).

What sets them apart from the parallel / clustered file systems is that they are deployed in medium-to-large clusters (≈ 1000 nodes accounted for `GFS` in [10], and ≈ 4000 nodes are claimed for HDFS by Yahoo) comprised of low-end off-the-shelf hardware, with normal rate-of-failure; plus these nodes not only are allowed to fail, but are expected to do so (due to the failure rate and the high number of nodes). Also each node doesn't need access to a shared storage medium, but only to locally attached one. Thus in this case the data availability is solved by replicating (dispersing) the data to other nodes. On the other side the disadvantages (which are outweighed by their scalability) are that even if the overall performance is good, the performance for a single file access is given by the individual characteristic of the low-end hardware.

The classical architecture for such a file system is the following [10] [20]:

- *master server (in `GFS` terminology), or `MDS` (Meta-Data Server) (in Ceph's one)* — a node or group of nodes in a master-slave replication with fail-over relation, which hold only information about what files are stored, where are their replicas stored, and other miscellaneous meta-data; these servers act as indirectors for clients and monitors for the data nodes described below; they are not expected to fail as much as the other ones; and their failure disrupts the entire operation of the file system;

- *chunk servers (in `GFS` terminology), or `ODS` (Object-Data Servers) (in Ceph's one)* — this is where the large number of nodes reside, and they hold the actual file data;

The classical file storage / retrieval algorithm is as follows: once a client decides it needs to access a file at a certain range, it contacts the `MDS` server and asks him the where-abouts of the `ODS` that stores that particular file, and more exactly that particular range of the file. The range information is critical in this lookup as one file (which could be very large) is not stored entirely on one server, but actually distributed throughout the entire cluster. Once the client obtains the location of the data node, it connects directly there and transfers data in or out. [10] [20]

Thus by analyzing the above situations (and the scientific literature), we can conclude that these distributed file systems are very different in nature than normal local or networked file systems, and thus we can't use them without taking this into account (the "mindshift"):

- because of the way the file access mechanism is implemented, it is better to store and access a small number of large files, and to improve as much as possible the locality of the data access;

- these file systems tend to be "eventually consistent" (more about this in the next section) which means that we need to explicitly control the access of different clients to the same data file;

- there is a performance / availability trade-off (again more about this in the next section) at least for the writing operations, as the data to be stored must reach to all the replicas; but on the other side there is a gain in read performance;

As a consequence of the above restrictions, many if not all cloud providers keep their distributed file systems private (Google's, or Amazon's example), and provide other higher level abstractions for such a resource (described in the next section). Meanwhile others (like Hadoop's `HDFS`) allow direct access to the file system but the applications need to use specialized API's in order to access it (in Hadoop's particular case there are also other limitations which further discourage the raw usage).

Thus we must keep in mind that although powerful these file systems are difficult to use and manage, and we must avoid as much as possible their direct usage. (They could be regarded as raw block disks are seen in today's operating systems.) As such I direct my attention to distributed databases (described in the next section), which are more often and heavily used in cloud applications than the distributed systems. (Also the proliferation of distributed databases outweigh that of the distributed file-systems, and such the amount of available information and implementations is in their advantage.)

2.3.2 Distributed databases

Again like in the case of distributed file systems we must make a distinction between *"real cloud enabled databases"*, and other kinds of databases which use multiple nodes:

- *replicated databases* — which are situated in the lower part of the scalability axis, because their main purpose is to replicate the same data on all the nodes — both from an availability concern, but also to optimize read operations; unfortunately their scalability reaches the limit when any node reaches the limit of how much data it can manage locally;
- *partitioned / sharded databases* — where although the data is distributed amongst different servers, their dynamism is quite constrained as adding new nodes in the cluster means repartitioning the entire data space; also another limitation is related with the failure model, as if a node fails, the entire data that was stored there becomes unavailable;
- *distributed databases* — that were designed from the beginning with dynamic node lifecycles and failures; and like in the case of distributed file systems vs networked file systems, these distributed databases are either constructed to leverage those file systems, or to work in a similar context with on off-the-shelf low-end hardware;

The pioneers in this area were again Google with its *"BigTable"* [5] and Amazon with its *"Dynamo"* [7], and after that many clones started to appear, the most prominent ones being: HBase (part of the Hadoop framework), Hypertable (sponsored by the Chinese search-engine Baidu), Cassandra (open sourced by Facebook), Riak (developed by Basho), just to name a few. And in order to better understand these databases — their data models, appropriate usages, etc. — we have to split them in three disjoint categories, and discuss them in isolation: columnar stores, key value stores, and document stores.

But before delving into that subject we have to first understand a key theoretical concept: *"the `CAP` (Consistency Availability and Partitioning) theorem"*, and the *"eventual consistency"* that stems as a consequence.

Eventual consistency

We shall first start with this topic as it is more simple to grasp, and it constitutes the first *"hard"* example that we cannot apply common techniques within cloud computing. Although not (thoroughly) formally described in a scientific paper (the closest being an two articles published in Communications of ACM [19] and [16]) eventual consistency is a corner stone in understanding these distributed databases. It starts by describing three main levels of consistency in any distributed database:

- *strong consistency* — if we execute a read operation after a previous write operation (for the same key), we shall obtain exactly the same result regardless of the node from which we are executing the operations; (this is usually what we would get out of a normal / clustered relational database;)
- *weak consistency* — the same as above happens, but only after a certain set of constraints is imposed (e.g. using transactions, waiting a specified amount of time, etc.);
- *eventual consistency* —

  > The storage system guarantees that if no new updates are made to the object eventually (after the inconsistency window closes) all accesses will return the last updated value. The most popular system that implements eventual consistency is DNS, the domain name system. Updates to a name are distributed according to a configured pattern and in combination with time controlled caches, eventually of client will see the update.

Thus the first lesson we take out is that all we know from relational database theory related to isolation levels and their guarantees disappear — not even the *"read uncommitted"* isolation level applies, as we might not even see what another process has written on another database node — and as a consequence we must rethink our applications with these new constraints in mind. However there are a few solutions that we could apply:

- *synchronization* — each writing process should synchronize its access by external (not provided by the database) methods (e.g. distributed lock managers, centralized coordinator, etc.); unfortunately this solution is not acceptable as it negates almost all benefits enabled by the distributed database (especially scalability and availability);
- *ignorance* — using these databases only for non-critical records, where by overwriting a concurrent write operation does not incur losses or costs; (e.g. distributed caches, volatile / frequently changing environmental parameters, etc.;)
- *append only* — by which instead of overwriting any records, we keep creating new records (thus obtaining a logical append operation); of course there is also a need of a *"garbage-collection"* process that cleans or merges old records (and for which the same inconsistencies constraints and solutions apply);
- *versioning* — (another form of append-only) by delegating to the distributed database the task to store and retrieve multiple versions of the same logical record, and leave the merging decision to the actual implementation;
- *parametrization* — whereby the application has the choice to specify the consistency level per each operation;

`CAP` Theorem

Before getting any further, we should also pay attention to the conjecture [4] (and then proved theorem [11]) which lays at the foundation of the above described limitations — the `CAP` (*Consistency–Availability–Partitioning*) conjecture / theorem:

> You can have at most two of these properties [consistency, availability, (tolerance of network) partitioning] for any shared-data system.

Taken one by one these properties mean:

- *consistency* — relates to *"strong consistency"* described earlier;
- *availability* — implies that whatever (non-catastrophic, and non-global) event happens to the distributed database nodes (their interconnections, storage, operating system, or application itself), the overall system (as seen from outside the database system itself), shall not become unavailable (i.e. unresponsive or returning error responses); translating this into our context means that no matter what, the system should either give a previous (older) version of the data (if such exists), but under no circumstances should it deny read or write access;
- *partitioning* — (referring to network partitioning) implies that in the case of a complete split of the database in two or more partitions (groups of nodes, for which the nodes in one partition can communicate with any other node in the same partition, but there is no / limited communication with nodes in other partitions) no disruption of service is to be encountered — we are able request any operation to any node from any partition; again the translation to our context is the same as above;

Thus as the conjecture / theorem states, we can have only two of these properties, we have the following implementations:

- *consistency + availability (– partitioning)* — BigTable, HBase, Hypertable;
- *availability + partitioning (– consistency)* — Dynamo, Riak, Cassandra;
- *consistency + partitioning (– availability)* — most clustered / replicated databases, which in case of partitioning become read-only;
- *configurable* — (allowing the user to select which of the two properties are important (or at least a degree of such importance) Dynamo, Riak, Cassandra;

Model categories

Another characteristic of these distributed databases that differentiate them is the data model — the way in which logical records are structured, stored, accessed or queried. Thus in what follows I shall describe the three main categories: *"columnar"*, *"key-value"* and *"document"* databases.

Columnar databases resembling — at least superficially — with the relational model: each record has a unique key read-only (*"primary key"* in relational language), then for each column family (somehow resembling *"table spaces"* or *"columns"* from relational databases) we can store one or more columns (not to be confused with *"columns"* in relational model). In fact we can see these columnar stores as a big hash-tables, where each record key has associated another hash-table (with fixed keys), the column families, and for each column family we have another hash-table containing pairs of columns and values. Examples of such databases are: BigTable, Hypertable, HBase, Cassandra. (The model is best described in [5].)

Key-value databases which are a 1:1 match of hash-tables or sorted trees, associating for an arbitrary key an arbitrary binary value. Examples are: Dynamo / S3, Riak, Voldemort, Membase, KumoFS, etc. (The model is best described in [7] or [9].)

Document databases resembling `XML` databases where we store unstructured `XML` or `JSON`, and then we can query them with special purpose languages. Examples are CouchDB (not actually a distributed database, but it was the first one), MongoDB (providing only sharding).

But regardless of the data model they all share the same characteristics / limitations / constraints:

- *denormalization* — contrary to the relational theory, the data is stored in an denormalized fashion, for two reasons: firstly because there are no indices, thus querying for anything else than the record key (the primary key in relational world) would lead to a full scan of the entire distributed data;
- *limited range queries* — another severe limitation of most distributed database implementation is the lack of any range query support (i.e. requesting all records with a key between two values); for example the only ones that currently support such an operation are BigTable, Hypertable, HBase, and (with some implications) Cassandra (all these being columnar databases); no distributed key-value or document database (to my knowledge?) allows for such queries;

- *(native) lack of arbitrary queries or aggregation* — unlike in the relational model where filtering (`WHERE` keyword), or aggregation (`GROUP BY`, `AVG(...)`, etc.) are normal and often used operations, they lack completely in distributed databases; (granted that for example filtering can be obtained by manually modelling indices, or where known aggregates can be built and maintained by hand;)

2.3.3 Map-reduce

We cannot continue and discuss cloud computing without mentioning the famous *"map-reduce"* method [6]. It's working principle is trivial, but when distributed on a great number of nodes (and implemented properly), it can provide tremendous efficiency (it is applicable on most embarrassingly parallel applications that deal with high volumes of data):

- *partitioning* — the idea is that the large volumes of input data must be split (with minimum processing power) in smaller chunks, so that the number of chunks is at least an order of magnitude greater than the number of available nodes; (although a trivial and often underestimated step, it has tremendous effects on the overall performance, which I shall discuss later in this section;)
- *mapping* — within which each computing node receives a chunk of data which it has to process, and as an output we obtain new chunks identified by some arbitrary keys; the keys must be (usually sortable), or at least comparable for equality; in general there is a $n : n$ relation between the chunks and keys;
- *sorting* — where the previous outputted keyed chunks are sorted by keys and pushed to the next step;
- *reduce* — where each node is assigned a set of keys that fall under its responsibility; thus each node receives all the keyed chunks that have the same key as the node, and it aggregates it in an application dependent way; outputting a final chunk of data;

As seen from the previous description it seems a very simple process, but there are a few hidden characteristics that give this method its versatility, and at the same time constrain what can / should be done at each step:

- *parallelization* — although the description seems sequential (first partition, then map, then reduce), they are actually executed in parallel; thus after the first chunk exits from the partitioning process, it can be served to the first mapping node; and as soon as the first keyed chunks exits it can be forwarded to the reducing node; (indeed there is the restriction that once a reduce node is assigned a key it must keep this assignment; and because of this there is asserted that the keys from the mapping phase are uniformly distributed;
- *composability* — it must be noted that the reduce step can in its turn output keyed chunks, that in a similar fashion can be fed into another layer of reduce processors, thus obtaining a multi-step map-reduce process;
- *input data locality* — because usually the input data is stored in distributed file systems or databases, when processed by the mapping phase, the data movement must be minimized, thus the importance of the partitioning scheme, which must take into account two factors: where is the input data actually located, and where are the processing nodes located, and thus trying to minimize overall data movement costs;

- *support data locality / minimization* — during some specific mapping processes there is need for additional data — e.g. in the case of search engine indexing, and assuming that a mapping chunk represents the contents of a web page, in order to apply data-mining techniques (like stemming, etc.) we must know some meta-information about the web-site (like language, encoding, etc.) which could be located in an external database and not embedded in the chunk itself — such external additional data should either be kept to a minimum, cached locally, and in general it helps if all this external data is found in (predominantly) the same place (i.e. the same distributed database node);

- *side effects* — because computing nodes are expected to fail all map-reduce frameworks re-execute failed tasks on nodes that are available; thus our processing must not update any external data, or in case it does we must be prepared to encounter such double updates; (even worst Google's map-reduce implementation — in order to reduce the total execution time — it executes the last chunks multiple times on the machines that are unused, thus a chunk is considered completed when the first node completes;)

Consequently though simple in principle map-reduce is non trivial to put in practice, and we must pay attention to all these *"fine-script"* details that could negate all the benefits. As such the production ready implementations are quite a few: Google's map-reduce (not available to the general public) or Hadoop's map-reduce implementation. (There are other simpler / special-purpose implementations like Disco (although generic it is still young) or Riak's map-reduce (allowing only aggregation operations on their key-value database).)

2.3.4 Asynchronous communication

As said in the beginning of this work, very few papers pay attention to the communication patterns of cloud computing applications. But if we study the available (unstructured / uncertified) Internet sources (e.g. [18], [1], or [14]), we can conclude that most of the scalable / flexible cloud (or cloud enabled) applications follow the same pattern:

- *low-latency front-ends* — where they accept requests, execute the ones that are immediate, and delegate the ones that take longer to other modules through the use of queues;

- *heavy lifting back-ends* — that take out jobs from the before mentioned queues execute them, and maybe further delegating work in the same manner;

- *scalability* — if the back-ends can't face the incoming wave of tasks, the system automatically spawns new ones until the queues are under control;

- *fault tolerant* — either provided by the queues themselves, or emulated by the back-ends, only after a task has been finalized does the back-end actually acknowledge it, thus if a back-end breakage is encountered, or it times out; the task can be re-assigned to another back-end;

Although what has been described here resembles a lot with classical batch processing systems (Condor, PBS, etc.), its conceptually different:

- *smaller granularity* — in classical batch processing a job is atomic – everything from input to effective result — happens on the same assigned node, within the same job; meanwhile in the case of cloud computing applications these tasks resemble closer to asynchronous requests, where each tasks represents an unitary piece of work, which in turn may generate other tasks;

- *dynamic workflows* — we can argue that these tasks generating tasks represent actually an instance of a workflow (which some batch processing systems support, e.g. DagMan for Condor, or which can be handled by BPEL engines); but they are set apart by the fact that the resulting workflows are dynamic in nature, the processing steps being determined only at run-time; also on the other hand the available workflow engines incur great overhead and induce high latencies;

As exponents of this technology we have Amazon's SQS (*Simple Queueing System*) or SNS (*Simple Notification System*) [9] and the recently proposed industry standard AMQP (*Advanced Message Queuing Protocol*) [12] best implemented by RabbitMQ.

On the scientific side we also have a paper [21] (and a PhD. thesis) on a related subject — that instead of proposing message exchange between independent (i.e. not in the same process space) components — it focuses on applying this technique of using queues and tasks inside high-performance multi-threaded servers (thus obtaining higher processing capabilities than with classical multi-threaded or multi-processor systems).

2.4 Cloud mindset

As I have stated in the beginning of this chapter — and hopefully convincingly argued in the previous sections — it doesn't only suffice to replace physical hardware with virtualized machines, or to replace relational databases with columnar ones, but because new rules apply (i.e. constraints, limitations, pitfalls, etc.) — which pushes us to substantially change the way we design and implement cloud enabled applications.

Then new rules are:

- *decoupling* — (although not cloud specific, but a common sense one, this rule is extremely often broken) splitting our applications into independent components in the effort to minimize the dependencies and interactions with other components; and where such interfaces are identified they must be carefully designed as not to expose much of the internal structure or working of the components;
- *built-to-fail* — no matter what the `SLA` promises there will be breakages — there are scientific papers and uncounted Internet background noise covering this topic — and no matter how good programmers we are there are lurking bugs (the better and clever we are as developers the more difficult to reproduce and debug will be), therefore our applications must be built with failure in mind; (e.g. we must assume that accessing a data store will likely timeout and thus we must be prepared to connect to a different node (usually this is covered by most libraries); we also must assume that there is a possibility for a machine to be unexpectedly killed, thus we must be prepared to switch to another one, and spawn a new one instead; etc.)
- *asynchronous communication* — allowing us to simplify and decouple our components, and at the same time making the system more robust; (we must take special care that some queue providers (like Amazon) don't guarantee once-delivery;)
- *eventual consistency* — as with any distributed system it is impossible to have an instantaneous view of the entire system (without stopping it first in order to observe it); thus we must be aware when we can overwrite unread data, or in other cases how to correlate conflicting versions of the same record;)
- *life-cycle management* — more than before we need to fully automate all life-cycle tasks: deployment and decommission must be automatized (at least the execution part), and in order to have a glimpse of the overall performance of our application (as we scale to tens or hundreds of nodes) individual node metrics won't be enough, thus we need aggregated data and anomalies detection;)
- *autonomy* — (based on the previous item) this is where cloud computing crosses roads with another research field *"autonomous systems"*, in that our applications must be able to take informed and automatic decisions in order to serve their customers at optimum efficiency; but at the same time we must carefully embed this intelligence as we have to balance between quality-of-service offered to our customers, and the budget that is allocated to our application;
- *appropriate data model* — choosing the correct data model and implementation can make a huge difference in the overall performance, as each one is not generally suitable for any task; (this unlike with classical relational databases, where the major differences were either syntactical or performance related;)

Chapter 3

Research direction

After having presented the most important technologies that cloud computing has to offer, seeing their versatility and coverage, and together with the fact that a lot of teams are researching this field, it may seem that there is nothing new left to be done. And in some sense it might be true as almost all topics are covered, from virtualization, to data dispersal.

But luckily there are two niches still to be (fully?) uncovered:

- *cloud `SDKs` (Software Development Kit)* — that actually enable cloud clients to develop and deploy their applications on-top of the virtualized infrastructure;
- *asynchronous (queues) messaging exploration* — although asynchronous messaging is covered by other research fields (like multi-agent systems, networking, web services, etc.) I believe their full potential was not yet completely covered;

Cloud `SDK` As stated throughout this work, we can't just jump into cloud computing without preparing for such a leap, as I've tried to prove that we need a *"mindshift"* in the way we architect and implement our applications. Thus his change needs to happen at two different levels: architecture and development. At the architecture level there are already publications [2] that have started investigating this topic by proposing patterns and guidelines. Unfortunately at the engineering end — close to the developer — even though there are some projects or prototypes they either fail at either genericity — focusing on a certain development environment, like `GAE` (*Google App Engine*) (Java and Python), AppScale (a `GAE` clone for Java), etc. — either being locked for a single cloud provider — like `GAE` (,) or Joyent (JavaScript), etc.

Thus I see an opportunity in building such an open cloud `SDK` which I shall consider complete if it meets all the following requirements (ordered by priority):

- scalable
- generic
- platform independent
- development environment independent
- well engineered
- efficient
- developer centered

As such all that I have proposed to realize will be done as part of the `mOSAIC` (*Open-Source API and Platform for Multiple Clouds*) project and in collaboration with the local `IeAT` (*Institute e-Austria*) team or partners.

Bibliography

[1] Building scalable, reliable amazon ec2 applications with amazon sqs. http://sqs-public-images.s3.amazonaws.com/Building_Scalabale_EC2_applications_with_SQS2.pdf.

[2] Open cloud manifesto. http://www.opencloudmanifesto.org/, 2010. (accessed 28-September-2010).

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

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

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[8] Jeremy Elson and Jon Howell. Handling flash crowds from your garage. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association.

[9] Simson L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3 and sqs. Technical report.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

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

[12] AMQP Working Group. Amqp 1.0 revision 0 – recommendation. http://www.amqp.org/, 2010.

[13] David Hilley and David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings, 2009.

[14] Steve Huffman. Seven lessons on scalability from reddit. http://www.slideshare.net/carsonified/steve-huffman-lessons-learned-while-at-redditcom.

[15] Peter Mell and Tim Grance. The nist definition of cloud computing. Technical report, July 2009.

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

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

[18] Jinesh Varia. Cloud architectures. http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf.

[19] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[20] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[21] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.