Institute of Architecture of Application Systems

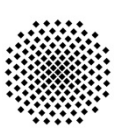
CAP-Oriented Design for Cloud-Native Applications

Vasilios Andrikopoulos, Steve Strauch, Christoph Fehling, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

BibTeX:

```
@inbook{ASFL2013,
  author    = {Vasilios Andrikopoulos and Steve Strauch and Christoph Fehling
               and Frank Leymann},
  title     = {CAP-Oriented Design for Cloud-Native Applications},
  booktitle = {Cloud Computing and Services Science},
  year      = {2013},
  pages     = {215--229},
  doi       = {10.1007/978-3-319-04519-1_14},
  series    = {Communications in Computer and Information Science},
  volume    = {367},
  publisher = {Springer International Publishing}
}
```

University of Stuttgart
Germany

CAP-Oriented Design for Cloud-native Applications

Vasilios Andrikopoulos, Steve Strauch, Christoph Fehling, and Frank Leymann

Institute of Architecture of Application Systems (IAAS), University of Stuttgart,
Universitätsstr. 38, 70569 Stuttgart, Germany

Abstract Brewer's conjecture, and its resulting formalization as the CAP theorem, impose serious limitations on the consistency, availability and network partitioning tolerance characteristics of distributed systems. Despite its importance however, few works explicitly consider the implications of the CAP theorem in the design of applications, especially for applications that are designed natively for the Cloud. In order to address this need, in this work we propose a CAP-oriented design methodology for Cloud-native applications. For this purpose we build and extend our previous work on Cloud architectural patterns. Finally, we show how the methodology can be used in practice to design an application solution with desired CAP properties.

1 Introduction

Cloud computing has been heralded as the realization of John McCarthy's utility computing vision, where computing is organized and offered as a public utility like electricity and water [1]. Cloud computing allows enterprises to outsource applications, systems and even their IT infrastructure to the Cloud, using one or more of the provisioned infrastructure or software services. Amazon, for example, offers Cloud solutions with usage-based costing, where interested parties can install and run their software without having to care about previously critical issues like infrastructure investment, computing power and network connectivity [2]. Salesforce.com altered radically the enterprise computing landscape by offering customizable services on the Cloud which were traditionally embedded in the IT domain of the enterprise. Cloud computing has ushered a new era of consuming and producing information and information technology by migrating the processing and storage of the information from small scale, limited purpose computing platforms like PCs, laptops and server machines to large scale, general purpose platforms offered "somewhere on the Cloud". This created the notion of *Cloud-native applications*, i.e., applications that are specifically designed and developed on top of a constellation of Cloud services, and which can fully exploit the characteristics of Cloud computing, e.g., elasticity [3].

Despite its revolutionary nature however, Cloud computing is underpinned by the same fundamental principles and laws governing large, distributed networked systems. One of the most important principles is a conjecture that Eric Brewer

put forward in his keynote speech at the ACM Symposium on the Principles of Distributed Computing (PODC) in 2000 [4]. Brewer observed that there are three fundamental systemic requirements in any distributed environment that exist in a special relationship with each other: consistency (whether all parts of the system see the same data at the same time), availability (what percentage of time the system is up and functioning properly) and network partitioning (if the system is tolerant to network failures). His conjecture is that only two out of these three requirements can actually be satisfied at any time by a distributed system. This hypothesis was later formally proven by Seth Gilbert and Nancy Lynch of MIT [5], making it known as the CAP theorem (from the initials Consistency, Availability and network Partitioning).

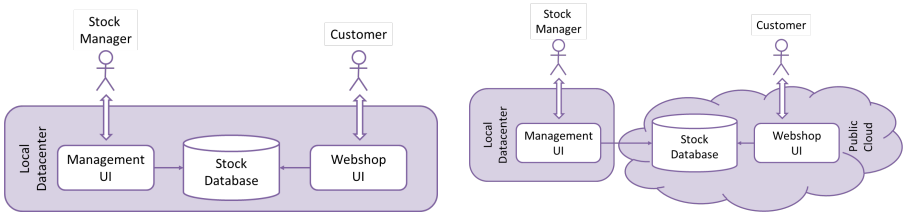
By its definition, the CAP theorem is restricting the capacity of any distributed system to satisfy requirements related to the CAP properties, and as such it has a direct impact on these requirements. This impact is even bigger for Cloud-native applications where elasticity, i.e., being able to deal with shifting computational demands by scaling up or down accordingly, is one of the basic pillars of the paradigm. Elastic applications should be able to maintain similar (or better) CAP behavior independent of their scale and rely on their design to do so. Studying and analyzing therefore the effect of various architectural decisions on the behavior of the resulting application with respect to the CAP theorem becomes an important issue and is the proposed goal of this work.

More specifically, in the following we present a design methodology for Cloud-native applications which is oriented towards connecting design decisions with an estimation of the CAP behaviour of the resulting application. Furthermore, we show how the methodology can be realized as an extension of the Cloud Pattern Framework presented in [6]. Finally, we validate our proposal using a scenario running through the paper.

The rest of this work is structured as follows: Section 2 motivates the need for a CAP-oriented design methodology by means of an example. Section 3 discusses the CAP theorem in more detail and presents the proposed application design methodology. Section 4 shows how the methodology can be realized in practice, while Section 5 discusses validation. Finally, Section 6 summarizes the related work, before providing some conclusions and possible future directions in Section 7.

2 Motivation

For illustrative purposes, consider the familiar example of a simple Web shop application as depicted in Fig. 1a. Customers browse through offered items using the Web shop user interface (*Webshop UI*). If they decide to order an item, it is packaged and sent to them by one of the stock managers in the shop using a management interface (*Management UI*). Both user interfaces access a common data store (*Stock Database*) containing the item descriptions and their availability. The complete Web shop is hosted on a local data centre, belonging to the shop owner. The Web shop, however, experiences very high workloads



(a) On premises (b) Off premises

Figure 1: Web shop example

during specific times of the year, for example, when Christmas approaches. The shop owner therefore decides to use elastic Cloud resources to cope with such alternating workloads.

Consulting online resources, he decides to completely outsource his data store and shop interface to the Cloud, where he can use the elasticity and scalability offered by it. He decides however not to outsource the management interface and continues hosting it on premises. The new architecture of the Web shop is shown in Fig. 1b. While the new Web shop fulfils the expectations in terms of computational resources in periods of increased activity, the owner is very quickly faced with a new problem: fulfilling the orders depends on the link between the management interface and the data store on the Cloud. Frequent network failures in this link force the stock managers to wait before processing an order, essentially creating a bottleneck in the application. In the following sections we are going to discuss how the shop owner (or more specifically, the application designer on his behalf) would have been able to foresee this problem before actually implementing the application.

3 CAP-Oriented Design

3.1 Design Decisions & CAP Properties

Since 2000 when Brewer posed his conjecture and until today, a number of works have appeared in the literature discussing the implications of the CAP theorem in system design, see for example [7], [8], [9], [10]. These discussions however stay on the level of particular cases and best practices and do not identify or organize the underlying principles of systems design for the Cloud. For purposes of visualization, it is more appropriate to think of the CAP properties positioned along the edges of a tetrahedron, as shown in Fig. 2, with minimum values for these properties in the intersection of the axes (marked with 0 in Fig. 2). CAP properties of different systems are positioned along the axes and form triangular areas that cut through this tetrahedron.

The strict interpretation of Brewer's theorem would position any system on one of the sides of the tetrahedron. In practice however, system designers and developers trade some degree of, e.g., consistency for availability and network

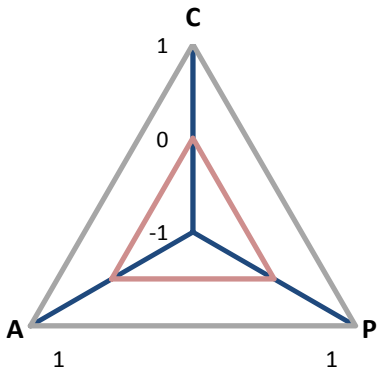


Figure 2: The CAP Properties of a Distributed System

partitioning. Proposed solutions like the one discussed in [11], where all three properties of CAP can be satisfied (not, however, at the same time), confirm that there is actually space to outmaneuver the constraints imposed by the CAP theorem with clever design. Systems like the Amazon.com online store, for example, allow customers to buy items without ensuring their physical availability at the time of purchasing. If, e.g., a copy of the requested book is not currently available in stock then it can either be purchased transparently to the customer through a third party, or the fulfillment of the order can be delayed until it becomes available (or ultimately some kind of compensation can be offered). The reasoning here is that customers should always be served, even in case of (internal to the systems of Amazon.com) network failures and even inventory inconsistencies. The consistency of the system will actually only be eventually ensured by a set of corrective actions [12]. Thus, in terms of Fig. 2, it can be said that the Amazon.com store is positioned closer to the A vertex. Other systems like for example online travel agencies, trade availability for consistency and network partitioning tolerance by making sure that no two customers book the same ticket, even in the presence of network failures. In this manner they essentially position themselves closer to the C-P side of the tetrahedron.

Different system requirements therefore lead to vastly different system design solutions, and different systems (in this case Cloud-native applications) end up in different areas of the tetrahedron in Fig. 2. Identifying the key decisions and their underlying principles, and connecting them with particular CAP properties is necessary for making sure that a Cloud-native application design fits its desired characteristics. Positioning the application in the tetrahedron is however not trivial. As demonstrated in the previous section, application design usually entails a series of architectural decisions, with each one of them having potentially a different effect on the CAP properties of the application. Furthermore, particular implementation decisions like, e.g., the choice of platform for hosting an application have an indirect effect on other decisions like the way the clients

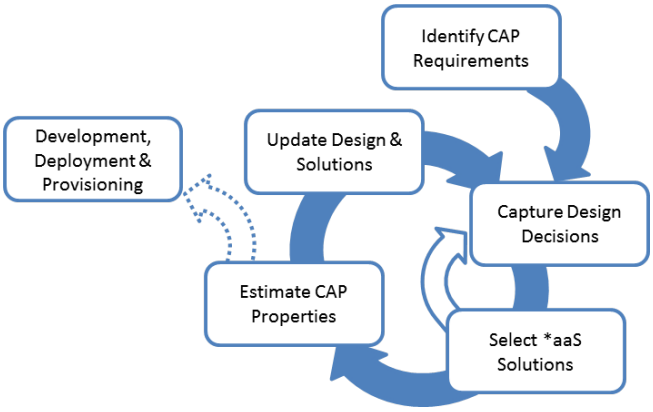


Figure 3: The CAP-oriented Cloud-native application design methodology

will access the application. Architectural decisions are therefore in a feedback loop and their effect for the CAP properties can only be estimated by taking into account their interplay dependencies.

3.2 Application design methodology

The CAP-oriented Cloud-native application design methodology presented here aims to address the requirements discussed above. It comprises of 5+1 phases, illustrated in Fig. 3 and presented in the following.

Identify CAP Requirements: The first phase requires of the application developer to identify the envisioned CAP properties of the designed application. For example, in the Web shop scenario discussed in the previous section, the migrated to the Cloud system requirements effectively call for stronger consistency, with network partitioning tolerance as a secondary goal, and availability only third. Actually positioning the desired outcome as a triangle in the tetrahedron of Fig. 2 provides the application designer with a qualitative feel of the requirements that he is building towards.

Capture Design Decisions: The second phase consists of recording the various decisions made by the application designer. This involves in the case of the Web shop scenario, the decision to use a public Cloud for hosting the application, the storage model chosen etc. Capturing these decisions (and indeed facilitating the design of the application) is better performed, as we will discuss in the following section, by means of a decision support system like the one discussed in [13] (see related work section for further information).

*Select *aaS Solutions:* The third phase of the methodology complements the previous phase by translating the various abstract design decisions into concrete

Software-, Platform- or Infrastructure-as-a-Service (*aaS) solutions. For the Web shop, for example, this may entail using the Amazon Web Services data storage solution. In principle, design decisions like the data storage model to be followed should "drive" the *aaS solution options. Choosing a particular solution however may influence previously taken design decisions with respect to its CAP properties. This may require a revisit of the previous phase, shown by the backward arrow in the loop of Fig. 3.

Estimate CAP properties: During this phase, the CAP properties of the various solutions are combined in order to provide an estimate of the overall CAP properties of the designed application. It is relatively easy to assume a binary nature of the properties following the strict interpretation of the theorem. However, as discussed by Brewer himself in [14], all properties are more continuous rather than binary. Different sub-systems also exhibit different properties and they contribute in different ways to the overall behavior of the system.

In order therefore to achieve an estimation of the CAP properties for the whole application, the selected *aaS solutions must be already annotated with information about their CAP properties. The annotation can be expressed as a triplet (c, a, p) with $c, a, p \in [-1, 1]$, where values closer to 1 signify a strong correlation with a property, while values close to -1 show a strong negative correlation, meaning that they affect this property of the application in a degrading manner. Estimating the properties of the system in this case can be performed by aggregating the various values for each property, and normalizing the result in the $[-1, 1]$ range. The advantage of this approach is that the result can be visualized in Fig. 2, which allows a designer to easily assess whether the designed application satisfies the requirements identified in the first phase. More sophisticated methods like log mining and stochastic methods can be used both for the actual extraction of the CAP properties of each *aaS solution and for their combination into one (c, a, p) triplet.

Update Design & Solutions: Based on whether the estimated CAP properties of the application satisfy its defined CAP requirements, the designer can choose either to proceed with the Development, Deployment and Provisioning of the actual application (not in the scope of this work), or re-enter the design cycle through the Update Design & Solutions phase. During this stage the designer attempts to identify and isolate the design decisions and *aaS solutions that produced the undesired outcome. Since changing any of them may have an impact on the overall design of the system, it is then required to re-enter the design decision/*aaS solution loop before estimating again the (new) CAP properties. This cycle may be repeated a number of times until a desired outcome is achieved.

4 Architectural Decisions & Design Patterns

In the previous section we presented a CAP-oriented design methodology for Cloud-native applications. The next step is to make this methodology concrete

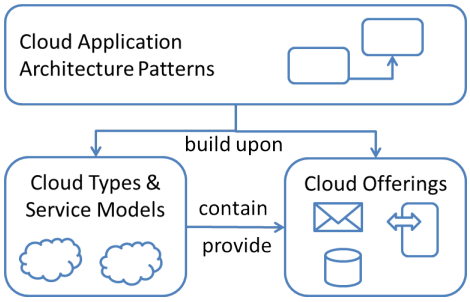


Figure 4: Pattern Classes

and demonstrate how it can be instantiated into a set of methods and tools for application design. For this purpose, in the following we focus on presenting the Cloud Pattern Framework introduced in [6], as the enabler of our methodology.

4.1 Cloud Architecture Patterns

Architectural patterns are used in many computer science domains to capture good solutions to reoccurring problems in an abstract common descriptive format, e.g., [15], [16]. A catalogue of patterns may then be used to guide application developers during the implementation. In our previous work, we abstracted the architectural principles of Cloud computing from existing Cloud applications and Cloud offerings and compiled them into a pattern catalogue [17], available also online at `http://Cloudcomputingpatterns.org`. In contrast to other pattern catalogues, we extend the use of the patterns to also describe the aspects of Cloud that are not implemented by the developer. This is necessary since Cloud applications rely heavily on runtime environments offered by Cloud providers. We describe the common concepts and behaviour of the environments in the same pattern format to ease their perception. This also allows the description of the environment in which a developer may apply Cloud architectural patterns through their interrelation to other patterns.

An overview of the resulting Cloud pattern classes is given in Fig. 4. Cloud Types & Service Models contain pattern-based descriptions of the Cloud environment. For example, there is a pattern for public Clouds (accessible by everyone), private Clouds (accessible within one company), community Clouds (accessible for a certain number of companies), and hybrid Clouds (a combination of at least two of the other types of Clouds) [3]. The Cloud environment that is described by this pattern class contains Cloud offerings providing computation, storage, and communication functionality. These Cloud offering patterns abstract from the concrete products of Cloud providers; for example, Amazon S3 or Windows Azure Storage are abstracted by the blob storage pattern. Architecture patterns may then be connected with these offering patterns to guide application developers when using these offerings.

Table 1: Excerpt from the Decision Recommendation Table [6]

	Public Cloud	Private Cloud	Community Cloud	Hybrid Cloud
Cloud Component Gateway	–	–	–	+
Elastic Infrastructure	+	+	+	+
Low-available Computer Node	+	∅	+	+
High-available Computer Node	∅	+	+	∅

Legend: +: strong relation –: exclusion ∅: no relation

4.2 Cloud Pattern Framework

To guide the application developer during the selection of applicable patterns for his concrete use case and Cloud environments, in [6] we introduced the Cloud Pattern Framework. In addition to the catalogue of patterns, a central component of the framework is a Decision and Solution Capturing component, enabled by a Decision Recommendation Table which captures the relations between the different patterns. We differentiate relations identifying the patterns to be (i) strongly related, (ii) mutually exclusive, and (ii) unrelated. Using this table (an excerpt of which is depicted in Table 1), an application developer iteratively selects patterns and receives recommendations for other patterns that may be applicable as well. Possible conflicts in the pattern selection can be identified through the evaluation of exclusion relations.

For example, an application developer may start by selecting patterns that describe the Cloud environment at hand for which the application is being developed. He selects the hybrid Cloud pattern in the decision recommendation table, because the application uses different Clouds for different application components. Based on this selection, the Cloud Component Gateway pattern is recommended to the developer. This pattern describes how application components may be made accessible in different Cloud environments in case of communication restrictions and has therefore a strong relation to the hybrid Cloud pattern. Navigating through the table in a similar manner from more higher-level to more low-level patterns (e.g., type of data storage or communication mechanisms) provides the designer with a set of choices for *aaS Solutions that implement the particular pattern. Other non-functional patterns, like for example the ones we discuss in [18] can also be used for this purpose.

At this point the designer can simply choose which solution to use for the application design. The actual guidance through the recommendation table, and the recording of the various decisions that were taken is performed by the Decision & Solution Capturing module, shown in Fig. 5. The Cloud Pattern Frame-

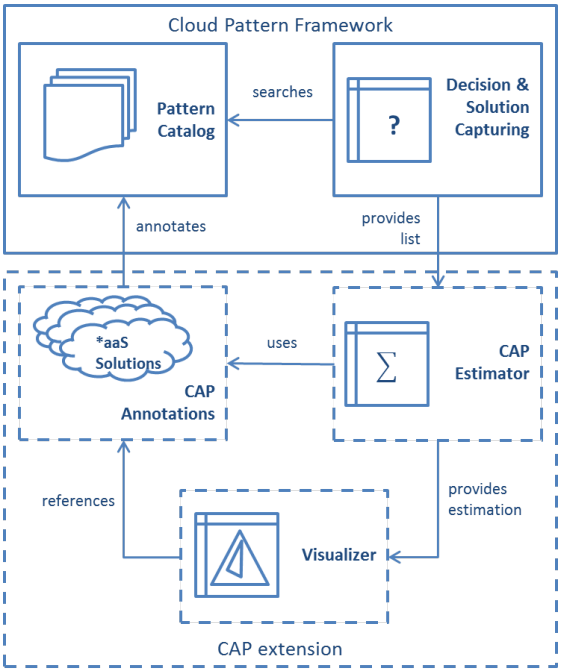


Figure 5: CAP Extension of the Cloud Pattern Framework

work therefore provides us with a set of useful building blocks (pattern catalogue, recommendation table, decision and solution capturing) for realizing the CAP-oriented application design methodology described in the previous section — as far as the decision capturing and *aaS solution selection phases of Fig. 3 are concerned. In the following we show how it can be augmented with CAP information in order to realize the Estimate CAP Properties phase.

4.3 CAP-Oriented Cloud Pattern Framework

In order to be able to estimate the CAP properties of an application in design we extend the Cloud Pattern Framework in three ways, as shown in Fig. 5 using dashed lines. More specifically, as a first step we annotate the *aaS Solutions contained in the Cloud Pattern Catalog with CAP Annotations. These annotations are triplets (c_i, a_i, p_i), where $c_i, a_i, p_i \in [-1, 1]$, in the manner discussed in Section 3.2. Currently, the triplets (c_i, a_i, p_i) are calculated by aggregating the values provided by different Cloud application developers by means of a questionnaire. The Amazon SimpleDB data storage service, for example, implementing the NoSQL Storage pattern, comes with two modes of operation: strict consistency (closer to traditional RDBMS) and eventual consistency. In the former mode, it is annotated with the triplet $(0.6, 0.25, 0.4)$, while in the latter with $(0.3, 0.75, 0.75)$. Similarly, providing a MySQL server as a Cloud offering (e.g.

being deployed inside a Windows VM in Windows Azure), and implementing the Relational Datastore pattern is annotated with $(0.95, 0.4, -0.25)$ since it is only marginally tolerant to network partitioning.

The actual values of the triplets are meant to provide a qualitative feeling of how strongly positive or negative CAP behaviour is exhibited by the *aaS solution, and they can only be interpreted in relation to each other. For example, the value $c_{MySQL} = 0.95$ stands for a solution much more oriented towards consistency than, e.g., $c_{SimpleDB_{Eventual}} = 0.3$. While currently these values are only aggregations of the opinions of a limited group of Cloud developers, in the future we plan to expose them to the users of the implementation of our proposed approach, and allow for providing their own perceived values. By these means we aim to be able to provide a more up-to-date annotation set which is in a feedback loop with its consumers. In addition, we shall be also able to allow designers to add annotations for systems that do not appear in the Pattern Catalogue, provided that they are first related to an appropriate pattern.

An alternative, more objective, approach would be to categorize *aaS solutions into *property classes* with fixed values for all solutions in the class. For purposes of availability for example, class 6 systems [19], meaning that they have 99% followed by four or more decimal nines availability, could be assigned $a = 1.0$, class 5 $a = 0.9$ and so on. Defining the values for the triplets in this case is reduced to creating the classification, deciding on the fixed values for the classes, and assigning each system to a class for each property. Both of the described approaches can also be combined, allowing to better reflect the expert knowledge.

For the second part of extending the Cloud Pattern Framework we focus on the providing a CAP Estimator (Fig. 5) module. The estimator takes as input from the Decision & Solution Capturing module the list of *aaS solutions already selected by the designer. It then retrieves the appropriate CAP annotations for these solutions and calculates the overall CAP triplet for the application:

$$(c, a, p)_{estimated} = (f_{i\geq 1}(c_i), f'_{i\geq 1}(a_i), f''_{i\geq 1}(p_i)). \tag{1}$$

Different functions f, f', f'' can be applied for the $(c, a, p)_{estimated}$ resulting in different interpretations of the expected behavior of the system. For example, a pessimistic approach would be to use $f_{i\geq 1}(a_i) = min_{i\geq 1}(a_i)$, signifying that the overall availability of the system is as good as its weakest link. More sophisticated functions can be used for this purpose, with different weighting for components and network links between them, for example. For purely illustrative purposes, in this work we use the average of each of the properties, as follows:

$$(c, a, p)_{estimated} = \frac{1}{n} \sum_{i=1}^{n} (c_i, a_i, p_i). \tag{2}$$

For a Cloud application for example that comprises a MySQL server installed inside a Windows VM on Azure (implementing the Relational Datastore pattern as we saw above) with annotation $(0.9, 0.7, -0.25)$ and a Management UI as a

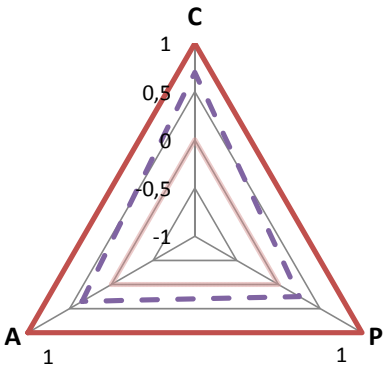


Figure 6: Visualization of the CAP Estimation

set of JSP pages on a local JBoss server (implementing the Stateless Component pattern) annotated with the triplet $(0.5, 0.0, 0.75)$ the estimated CAP properties are

$$(c, a, p)_{MySQL_{Azure}} = \frac{1}{2}(0.9 + 0.5, 0.7 + 0.0, 0.75 - 0.25) = (0.7, 0.35, 0.25).$$

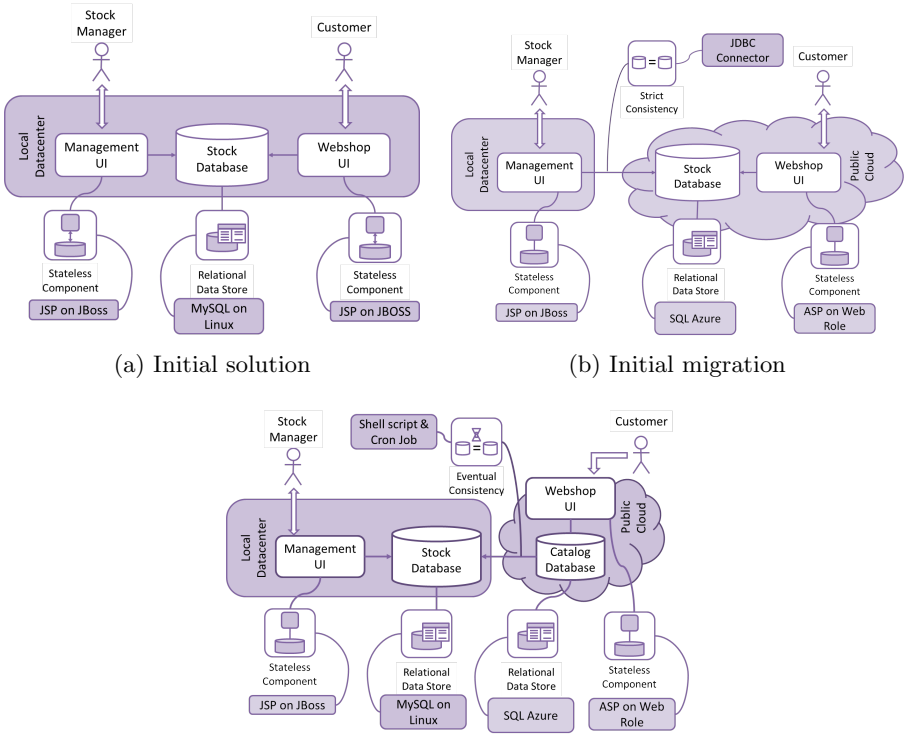
The estimated CAP properties show a system with high consistency but low availability and little tolerance in network partitioning (since it depends on the UI/Database link in order to operate correctly).

The visualization of this result is done by the Visualizer module in Fig. 5. The estimated CAP properties produced by the CAP Estimator are positioned as a triangle inside the CAP tetrahedron of Fig. 6 (extending that of Fig. 2). In the case of $(c, a, p)_{MySQL_{Azure}}$, the estimated CAP properties (illustrated by the dashed triangle) shows a clear tendency to the C vertex of the tetrahedron, denoting, as discussed above, strong consistency. The area bound by the lighter of the inner triangles in the centre of Fig. 6 denotes that one (or more) CAP properties of the application have a negative value.

Having extended the Cloud Pattern Framework to cater for the realization of the proposed CAP-oriented application design methodology, in the following we are going to validate our proposal by means of a case study. For this purpose we revisit the motivating scenario discussed in Section 2.

5 Case Study

Returning to the motivating example, the Web shop owner starts by annotating the current architecture with pattern information to determine the current CAP behaviour as depicted in Fig. 7a. Both user interfaces are Stateless Components (JSP pages on a JBoss server) relying on a Relational Datastore (MySQL on Linux), as external state. The links between them are synchronous and represent



(a) Initial solution

(b) Initial migration

(c) Migration with data replication

Figure 7: Web shop case study

data base queries and, therefore, have no pattern annotated to them. From the *aaS annotations catalog, we already know that $(c, a, p)_{JSP_{JBoss}} = (0.5, 0.0, 0.75)$ and $(c, a, p)_{MySQL_{Linux}} = (0.95, 0.4, 0.25)$. Therefore:

$$(c, a, p)_{Initial} = \frac{1}{3}(2 \times 0.5 + 0.95, 2 \times 0 + 0.4, 2 \times 0.75 + 0.25) = (0.65, 0.13, 0.58).$$

In a similar manner, and for the migration to the Cloud shown in Fig. 7b, we can see that $(c, a, p)_{Migration} = (0.68, 0.53, -0.14)$, since

$$(c, a, p)_{SQLAzure} = (0.75, 0.9, -0.5), (c, a, p)_{ASP_{WebRole}} = (0.5, 0.7, -0.3), \text{ and}$$
$$(c, a, p)_{JDBC} = (0.95, 0.5, -0.5).$$

The estimated CAP properties of the application reflect the observed ones in practice: much higher availability, roughly equivalent consistency, but very low partitioning tolerance (due to the stock management UI dependency on the availability of the communication link between the local data centre and the Cloud service). This result, and the relationship between the two application designs, is better illustrated in Fig. 8 where the exchange of network partitioning for availability is reflected by the positioning of the respective triangles.

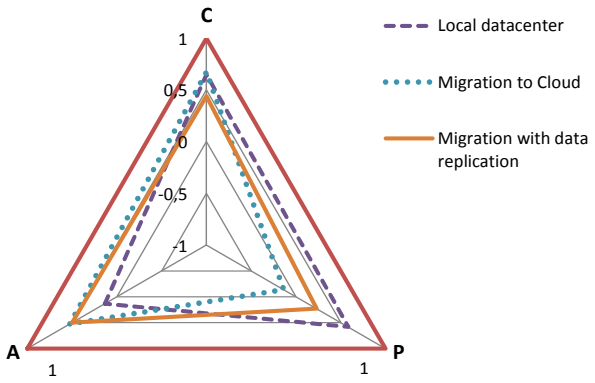


Figure 8: CAP Estimations for different Web shop Solutions

To ensure that the stock manager can work at all times, the shop owner decides to use the best of both worlds by replicating the data required by the stock manager and the customer as shown in Fig. 7c. The information required by the Web shop component is now contained in a separate catalogue component in the Cloud. The stock management component still contains all information about the goods and their availability. Hourly however, the data are replicated from the stock database to the catalogue database by a shell script and a cron job. This leads eventually to a consistency between the two data replicas as shown by the Eventual Consistency pattern annotated to the link. By calculating in a similar manner as above the estimated CAP properties, and for $(c, a, p)_{Script+Cron} = (-0.5, 0.5, 0.95)$, we have $(c, a, p)_{Replication} = (0.44, 0.5, 0.23)$.

This design solution therefore ensures that the availability is increased for both the stock manager and the customer and enables a system that is sufficiently partitioning tolerant by sacrificing a small amount of consistency: both the stock manager and the customer may access the information in the application, regardless of the availability of the communication link between the integrated runtime environments. The data consistency is however reduced, resulting in the possible condition that customers may order goods that are not available, because the actual product availability is only kept in the stock database. Therefore, compensation may be required in some cases, but the overall behaviour of the system is (probably) more profitable for the Web shops. Other Web shops like Amazon.com handle item availability in the same fashion. In all cases however, it is possible for the application designer to estimate the CAP properties by using the methodology and tools we discussed in the previous.

6 Related Work

Cloud application design (and engineering) is still a developing research topic, driven mostly by the industry. Solution providers like Microsoft, Amazon and

IBM have offered best practices on using their solutions for developing Cloud applications, see for example [2], [20], [21]. However, these are far from systematic software engineering approaches and they do not explicitly consider CAP properties. In a similar approach to ours, the work of [22] uses design patterns in Cloud application engineering. Their focus is on Cloud transformation, i.e., migrating existing applications to the Cloud.

Patterns are commonly used to describe good solutions to re-occurring problems in a common format to organize practical knowledge and ease perception. This concept has been used originally to describe building and city architecture [23] and has since been applied to a large variety of domains, such as learning [24] or business communications [25]. Regarding software architecture and runtime infrastructure, patterns have been defined for object oriented programming [16] and messaging-based application integrations [15]. Furthermore, different pattern catalogues capture good practices for user interaction with information [26]. These patterns have also been considered during the identification of Cloud computing patterns. Many of them were transformed or applied to the area of Cloud computing.

Capturing design decisions in order to focus and verify the design process of systems is also discussed in [13], where a formal model is presented for capturing and reusing architectural decision knowledge. Furthermore, in [27], the authors present a pattern-based approach for architectural decisions. Both approaches are conceptually close to this work, but discuss service-oriented and software systems and as such they are not directly applicable to Cloud-native applications. Further investigation on how they can be reused for this purpose is however in our future goals.

7 Conclusions & Future Work

While the CAP theorem has serious implications for the design of distributed systems (and therefore also of Cloud-native applications) there are few works discussing how to design for particular CAP properties. For this purpose, in this work we presented an approach for incorporating these properties into the design of Cloud-native applications. More specifically, we introduced a CAP-oriented design methodology which connects design decisions with existing Cloud solutions and provides the means to estimate the CAP properties of an application. This methodology was then realized by using Cloud patterns in order to capture the design decisions and a set of annotations on the various *aaS solutions that realize these patterns. A visualization approach was also presented that allows for better perception of the estimated CAP properties and their impact on the application design. Finally, the proposed approach was validated by means of a case study scenario.

In the future we plan to complete the annotation of the Cloud Pattern Catalog presented in [6] so that we can empirically validate our approach using different scenarios. As part of this effort, we also plan to extend the *aaS solutions annotation procedure to as large as possible group of Cloud experts and

offer tooling support for our methodology as an application in the Cloud. In addition we also plan to investigate different possible approaches in combining the CAP annotations, using for example weighted sums and other statistical methods. The proposed approach is geared towards building Cloud-native applications. The methodology discussed in Section 3, however, can be easily adapted and applied to the case of Cloud-enabled applications [28], i.e., applications that are partially or completely migrated to the Cloud. Depending on the selection of Cloud services to be used, and the envisioned topology of the migrated application, systems with radically different CAP properties could emerge. Combining this option with, for example, calculating the operational expenses of the migrated application in the Cloud, could result in a decision support system that would allow application stakeholders to figure out whether and how their application should be migrated to the Cloud.

Acknowledgements

The research leading to these results has partially received funding from the 4CaaSt project (`http://www.4caast.eu/`) from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 258862.

References

1. Leymann, F., Fritsch, D.: Cloud Computing: The Next Revolution in IT. Proceedings of the 52th Photogrammetric Week (2009) 3–12
2. Varia, J.: Architecting for the Cloud: Best Practices. Amazon Web Services (2010) `http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf`.
3. Badger, L., Grance, T., R., P.C., Voas, J.: Cloud Computing Synopsis and Recommendations. NIST Special Publication 800-146 (2012) `http://www.nist.gov/manuscript-publication-search.cfm?pub_id=911075`.
4. Brewer, E.A.: Towards Robust Distributed Systems. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing. Volume 19. (2000) 7–10
5. Gilbert, S., Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. ACM SIGACT News **33**(2) (2002) 51–59
6. Fehling, C., Leymann, F., Retter, R., Schumm, D., Schupeck, W.: An Architectural Pattern Language of Cloud-Based Applications. In: Proceedings of the Conference on Pattern Languages of Programs (PLoP). (2011)
7. Hewlett-Packard Development: There is no Free Lunch With Distributed Data. HP White Paper (2005) `ftp://ftp.compaq.com/pub/products/storageworks/whitepapers/5983-2544EN.pdf`.
8. Helland, P.: SOA and Newton's Universe. MSDN Blogs (2207) `http://blogs.msdn.com/b/pathelland/archive/2007/05/20/soa-and-newton-s-universe.aspx`.
9. Kossmann, D.: How new is the Cloud? In: Proceedings of ICDE 2010, IEEE (2010) 3–3
10. Mietzner, R., Fehling, C., Karastoyanova, D., Leymann, F.: Combining Horizontal and Vertical Composition of Services. In: Proceedings of SOCA 2010, IEEE (2010) 1–8

11. Pardon, G.: A CAP Solution (Proving Brewer Wrong). Personal Blog (2008) http://guysblogspot.blogspot.com/2008/09/cap-solution-proving-brewer-wrong.html.
12. Vogels, W.: Eventually Consistent. Communications of the ACM **52**(1) (2009) 40–44
13. Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N.: Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. Journal of Systems and Software **82**(8) (2009) 1249–1267
14. Brewer, E.: CAP Twelve Years Later: How the "Rules" Have Changed. Computer **45**(2) (2012) 23–29
15. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (2003)
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman (1994)
17. Fehling, C., Leymann, F., Mietzner, R., Schupeck, W.: A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-Based Application Architectures. Technical report, Technical Report (2011)
18. Strauch, S., Andrikopoulos, V., Breitenbücher, U., Kopp, O., Frank, L.: Non-Functional Data Layer Patterns for Cloud Applications. In: Proceedings of CloudCom'12, IEEE Computer Society Press (Dezember 2012) 601–605
19. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR (2000)
20. Erl, T., Kurtagic, A., Wilhelmsen, H.: Designing Services for Windows Azure. MSDN Magazine (2010) http://msdn.microsoft.com/en-us/magazine/ee335719.aspx.
21. Lau, C., B.V.: Best Practices to Architect Applications in the IBM Cloud. IBM DeveloperWorks (2011) http://www.ibm.com/developerworks/cloud/library/cl-cloudapppractices/index.html.
22. Chee, Y.M., Zhou, N., Meng, F.J., Bagheri, S., Zhong, P.: A Pattern-Based Approach to Cloud Transformation. In: Proceedings of CLOUD 2011, IEEE (2011) 388–395
23. Alexander, C., et al.: A Pattern Language. Towns, Buildings, Construction. Oxford University Press (1977)
24. Iba, T., Miyake, T., Naruse, M., Yotsumoto, N.: Learning Patterns: A Pattern Language for Active Learners. In: Conference on Pattern Languages of Programs (PLoP). (2009)
25. Manns, M.L., Rising, L.: Fearless Change: Patterns for Introducing new Ideas. Addison-Wesley Boston (2005)
26. Yahoo! Inc.: Yahoo! Design Pattern Library. Online Resource (2011) http://developer.yahoo.com/ypatterns/.
27. Harrison, N.B., Avgeriou, P., Zdun, U.: Using Patterns to Capture Architectural Decisions. Software, IEEE **24**(4) (2007) 38–45
28. Andrikopoulos, V., Binz, T., Leymann, F., Strauch, S.: How to Adapt Applications for the Cloud Environment. Computing (to appear) (2013) http://dx.doi.org/10.1007/s00607-012-0248-2.