Institute of Architecture of Application Systems

Designing for CAP - *The Effect of Design Decisions on the CAP Properties of Cloud-native Applications*

Vasilios Andrikopoulos, Christoph Fehling, Frank Leymann

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

BIBTEX:


```
@inproceedings{Andrikopoulos2012,
  author    = {Vasilios Andrikopoulos and Christoph Fehling and Frank Leymann},
  title     = {Designing for CAP - The Effect of Design Decisions on the CAP
Properties of Cloud-native Applications},
  booktitle = {Proceedings of the 2nd International Conference on Cloud
               Computing and Service Science, CLOSER 2012,
               18-21 April 2012, Porto, Portugal},
  year      = {2012},
  pages     = {365-374},
  publisher = {SciTePress}
}
```

These publication and contributions have been presented at
CLOSER 2012
CLOSER 2012 Web site: http://closer.scitevents.org

© 2012 SciTePress. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the SciTePress.

Universität Stuttgart
Germany

DESIGNING FOR CAP

The Effect of Design Decisions on the CAP Properties of Cloud-native Applications

Vasilios Andrikopoulos[1], Christoph Fehling[1] and Frank Leymann[1]

[1] *Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstr. 38, Stuttgart, Germany*

{vasilios.andrikopoulos,christoph.fehling,frank.leymann}@iaas.uni-stuttgart.de

Keywords: CAP Theorem, Cloud Patterns, Cloud-native Applications Design

Abstract: The limitations of distributed systems to satisfy the combination of consistency, availability and network partitioning tolerance requirements are well-documented and formally proven. There is however a limited amount of works discussing the impact of these limitations on designing applications native to the Cloud. This work addresses this particular need by proposing an approach for considering these requirements while designing an application. Our contributions are therefore a methodology for Cloud-native application design which considers consistency, availability and network partitioning tolerance, and a framework instantiating this methodology by using design patterns and their realization solutions on the Cloud. We also show how the proposed methodology can be used in practice to design an application solution with desired 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 (Leymann, 2009). 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.com, 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 (Varia, 2010). 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".

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 (Brewer, 2000). 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 (Gilbert, Lynch, 2002), 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 applications in the Cloud 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 behaviour

independent of their scale and rely on their design to do so. Studying and analyzing therefore the effect of various architectural decisions on the behaviour 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 (Fehling et al., 2011a). Finally, we validate our proposal using a scenario running through the paper.

The rest of the paper is 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 MOTIVATING EXAMPLE

For illustrative purposes, consider the familiar example of a simple Web shop application as depicted in Figure 1. 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 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 Figure 2. While the new Web shop fulfils the expectations in terms of computational resources in periods of

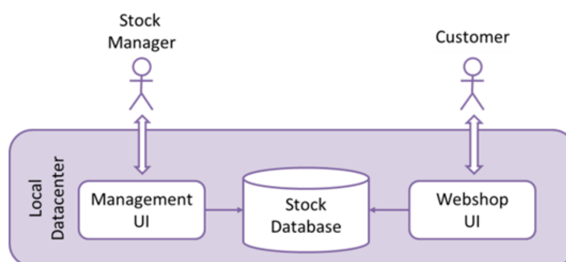


Figure 1 Web shop example: on premises architecture

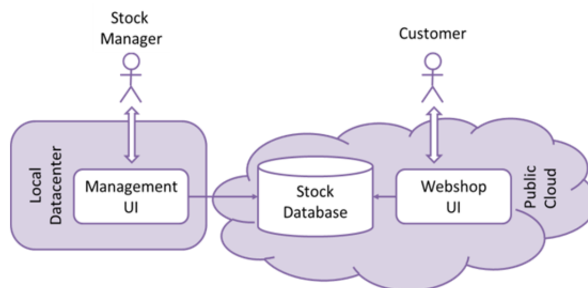


Figure 2 Web shop example: moving to the Cloud

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 (HP, 2005), (Helland, 2007), (Kossmann, 2010], (Mietzner et al., 2010). 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 as a triangle (see Figure 3), and the various systems

positioned within areas of this triangle. The strict interpretation of Brewer's theorem would position all systems on one of the sides of the triangle. In practice however, system designers and developers trade some degree of e.g., consistency for availability and network partitioning. Proposed solutions like the one discussed in (Pardon, 2008), where all three properties of CAP can be satisfied (not, however, at the same time), confirm that there is actually space to outmanoeuvre 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 fulfilment 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 (Vogels, 2009). Thus, in terms of Figure 3, it can be said that the Amazon.com store is positioned somewhere on the lower part of the triangle and closer to its 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 triangle.

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 triangle in Figure 3. 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 triangle 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 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.

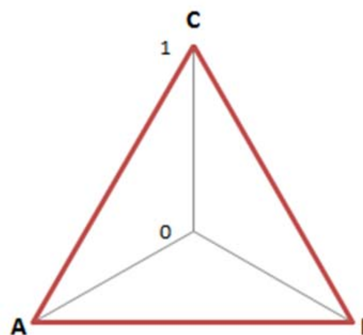


Figure 3 The CAP properties of a distributed system.

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 Figure 4 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 in the triangle of Figure 3 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 (Zimmerman et al., 2009) (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

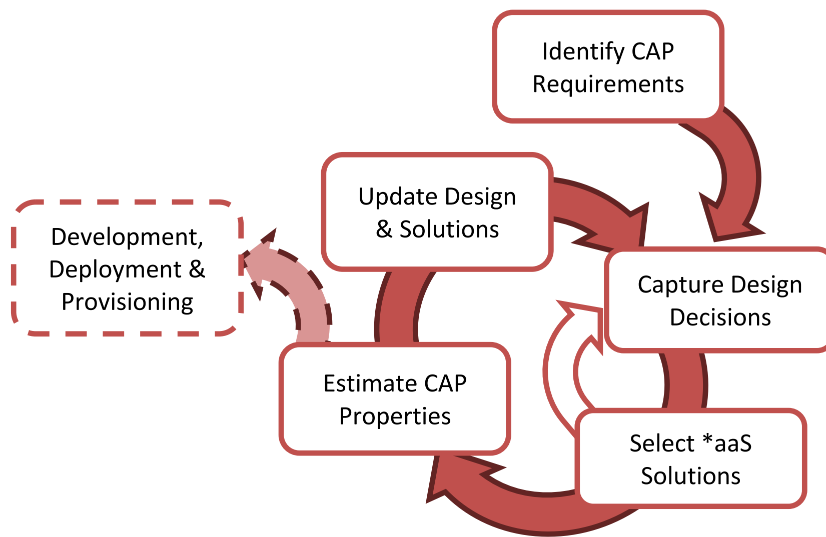


Figure 4 The CAP-oriented Cloud-native application design methodology.

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 Figure 4.

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. In order to achieve this estimation the selected *aaS solutions must be already annotated with information about their CAP properties. The annotation can be expressed for example 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. Estimating the properties of the system in this case can be performed by doing a weighted sum of the various values for each property, normalized in the [-1,1] range. The advantage of this approach is that the result can be visualized in Figure , 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.

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 CAPTURING DECISIONS THROUGH 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 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 (Fehling et al., 2011a), as the enabler of our methodology.

4.1 Cloud Application 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. (Hohpe, Woolf, 2004), (Gamma et al., 1995). 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 (Fehling et al., 2011b), 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 (Petre, 1995). This also allows the description of the environment in which a developer may apply cloud architectural patterns through their interrelation to other patterns by describing their cloud types and their offerings.

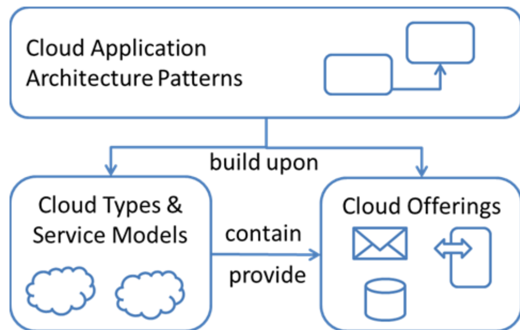


Figure 5 Pattern Classes in the Cloud Pattern Catalogue.

An overview of the resulting cloud pattern classes is given in Figure 5. 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. 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.

+ : strong relation – : exclusion Empty : no relation	Cloud Component Gateway	Elastic Infrastructure	Low-available Compute Node	High-available Compute Node
Public Cloud	–	+	+	
Private Cloud	–	+		+
Community Cloud	–	+	+	+
Hybrid Cloud	+	+	+	

Figure 6 Excerpt of the Decision Recommendation Table.

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 (Fehling et al., 2011a) 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 Figure 6), 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* (Fehling et al., 2011c) 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. 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 Figure 7. The Cloud Pattern Framework 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 Figure 4 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 Figure 7.

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. $c_{SimpleDB_Eventual} = 0.3$.

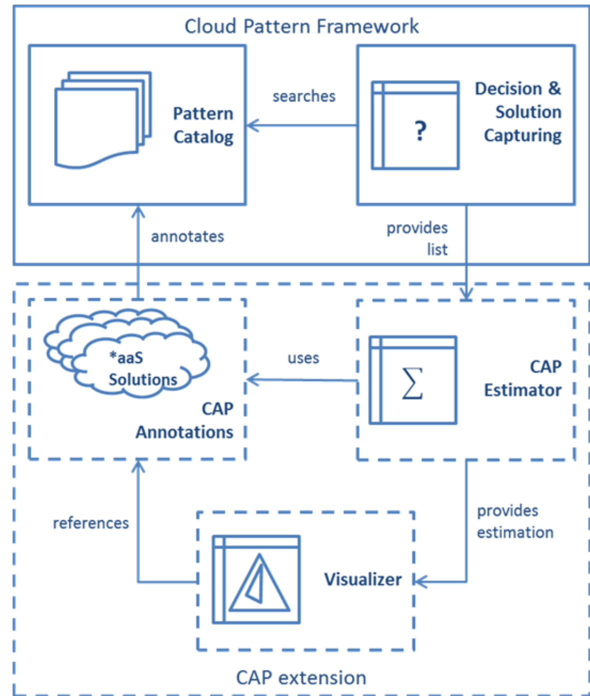


Figure 7 CAP extension of the Cloud Pattern Framework.

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.

For the second part of extending the Cloud Pattern Framework we focus on the providing a *CAP Estimator* (Figure 7) 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 by doing an average of each of the properties:

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

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 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

$$(c,a,p)_{MySQL_Azure}=$$

$$=\frac{1}{2}(0.9+0.5,\ 0.7+0.0,\ 0.75\text{-}0.25)$$

$$=(0.7,\ 0.35,\ 0.25)$$

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 Figure 7. The estimated CAP properties produced by the CAP Estimator are positioned in the CAP triangle of Figure 8 (extending that of Figure 3). 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 triangle, denoting, as discussed above, strong consistency. The area bound by the lighter of the inner triangles in the centre of Figure 8 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.

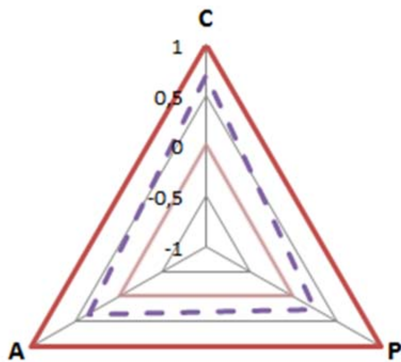
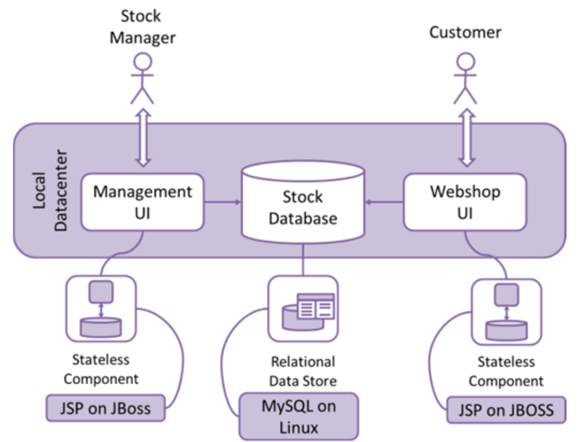
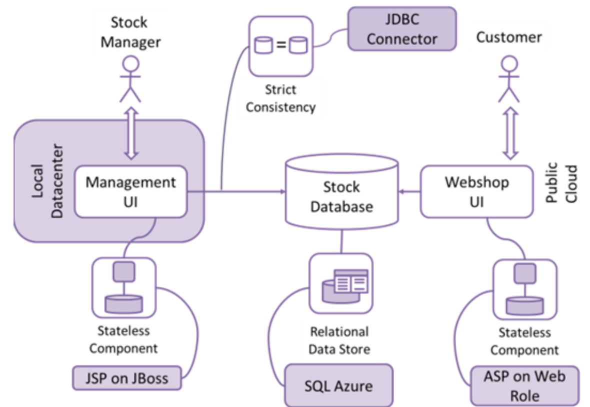


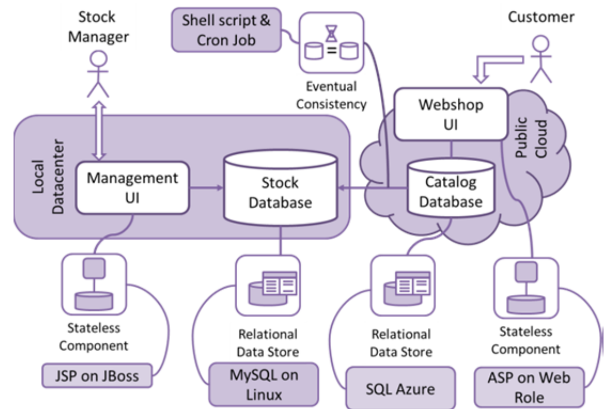
Figure 8 Visualization of the CAP estimation.



a. Initial solution



b. Initial migration to the Cloud



c. Migration with data replication

Figure 9 Web shop case study

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 Figure 9a. 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

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:

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

In a similar manner, and for the migration to the Cloud shown in Figure 9b, we can see that $(c,a,p)_{Migration} = (0.68, 0.53, -0.14)$, since $(c,a,p)_{SQLAzure} = (0.75, 0.9, -0.5)$, $(c,a,p)_{ASP_WebRole} = (0.5, 0.7, -0.3)$ 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). This result, and the relationship between the two application designs, is better illustrated in Figure 10 where the exchange of network partitioning for availability is reflected by the positioning of the respective triangles.

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 Figure 9c. 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

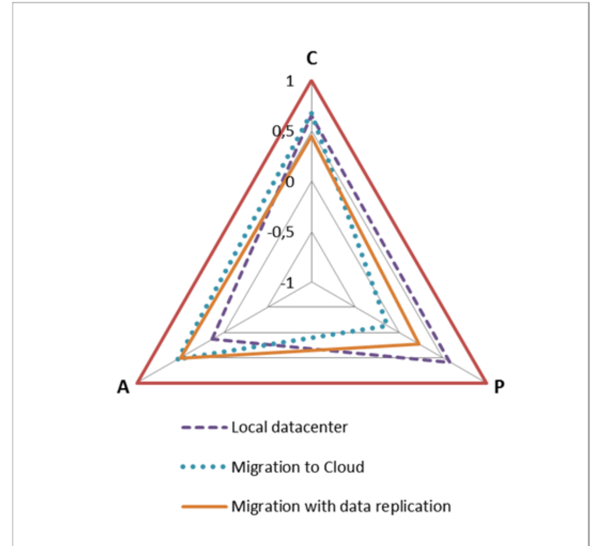


Figure 10 CAP estimations for different Web shop solutions

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 (Erl et al., 2010), (Varia, 2010), (Lau, Birsan, 2011). 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 (Chee et al., 2011) uses design patterns in cloud application engineering. However, their focus is on cloud transformation, i.e. migrating existing applications to the cloud, instead of designing Cloud-native applications.

Patterns are commonly used to describe good solutions to re-occurring problems in a common format to organize practical knowledge and ease perception (Petre, 1995). This concept has been used originally to describe building and city architecture

(Alexander et al., 1977) and has since been applied to a large variety of domains, such as learning (Iba et al., 2009) or business communications (Rising, 2004). Regarding software architecture and runtime infrastructure, patterns have been defined for object oriented programming (Gamma et al., 1995) and messaging-based application integrations (Hohpe, Woolf, 2004). Furthermore, different pattern catalogues capture good practices for user interaction with information (Tidwell, 1998), (Yahoo, 2011). 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 (Zimmermann et al., 2009), where a formal model is presented for capturing and reusing architectural decision knowledge. Furthermore, in (Harrison et al., 2007), 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

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 (Fehling et al., 2011a) 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. Finally we also intend to extend the methodology discussed in the previous for purposes other than CAP estimation, for example cost estimates and greenness of the application.

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. This paper expresses the opinions of the authors and not necessarily those of the European Commission. The European Commission is not liable for any use that may be made of the information contained in this paper.

REFERENCES

Alexander , C., Ishikawa, S., Silverstein, M., Jacobson, M., Fijksdal-King, I., 1977. A pattern language: towns, buildings, construction. Oxford University Press.

Brewer, E. A., 2000. Towards robust distributed systems. PODC Keynote.

Chee, Y., Zhou, N., Meng, F. J., Bagheri, S., Zhong, P., 2011. A Pattern-Based Approach to Cloud Transformation. Proceedings of the IEEE International Conference on Cloud Computing (CLOUD).

Erl, T., Kurtagic, A., Wilhelmsen, H., 2010. Designing Services for Windows Azure. MSDN Magazine. http://msdn.microsoft.com/en-us/magazine/ee335719.aspx

Fehling, C., Konrad, R., Leymann, F., Mietzner, R., Pauly, M., Schumm, D., 2011c. Flexible Process-based Applications in Hybrid Clouds. *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD).*

Fehling, C., Leymann, F., Retter, R., Schumm, D., Schupeck, W., 2011a. An Architectural Pattern Language of Cloud-based Applications. *Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP).*

Fehling, C., Leymann, F., Retter, R., Schumm, D., Schupeck, W., 2011b. A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-based Application Architectures. *Technical Report University of Stuttgart.*

Gamma E., Helm R., Johnson R., Vlissides J., 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley.

Gilbert, S., Lynch, N. A., 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33(2): 51-59*.

Harrison, N., Avgeriou, P., Zdun, U., 2007. Architecture Patterns as Mechanisms for Capturing Architectural Decisions, *IEEE Software*.

Helland, P., 2007. SOA and Newton's Universe. *MSDN Blogs.* http://blogs.msdn.com/b/pathelland/archive/2007/05/20/soa-and-newton-s-universe.aspx

Hewlett-Packard Development, 2005. There is no free lunch with distributed data. *HP White Paper*. ftp://ftp.compaq.com/pub/products/storageworks/whitepapers/5983-2544EN.pdf

Hohpe, G., Woolf, B., 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying.* Addison-Wesley.

Iba, T., Miyake, T., Naruse, M., Yotsumoto, N., 2009. Learning Patterns: A Pattern Language for Active Learners. *Proceedings of the International Conference on Pattern Languages of Programs (PLOP).*

Kossmann, D., 2010. How new is the cloud? *Proceedings of the IEEE 26th International Conference on Data Engineering (ICDE).*

Lau, C., Birsan, V., 2011. Best practices to architect applications in the IBM Cloud. *IBM Developer Works.* http://www.ibm.com/developerworks/cloud/library/cl-cloudapppractices/index.html (2011)

Leymann, F., 2009. Cloud Computing: The Next Revolution in IT. *Proceedings of the 52th Photogrammetric Week.* http://www.ifp.uni-stuttgart.de/publications/phowo09/010Leymann.pdf

Mietzner, R., Fehling, C., Karastoyanova, D., Leymann, F., 2010. Combining horizontal and vertical composition of services. *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA).*

Pardon, G., 2008: A CAP Solution (Proving Brewer Wrong). *Personal Blog.* http://guysblogspot.blogspot.com/2008/09/cap-solution-proving-brewer-wrong.html

Petre, M., 1995. *Why Looking isn't Always Seeing.* Communications of the ACM.

Rising, L., 2004. *Fearless Change: Patterns for Introducing New Ideas: Introducing Patterns to Organizations.* Addison-Wesley.

Tidwell, J., 1998. A Pattern Language for Human-Computer Interface Design. *Washington University Tech. Report WUCS-98-25.*

Varia, J., 2010. Architecting for the Cloud: Best practices – Amazon Web Services. *Amazon.com white paper.* http://jineshvaria.s3.amazonaws.com/public/cloudbestpractices-jvaria.pdf

Vogels, W., 2009. Eventually consistent. *Communications of the ACM 52(1): 40-44.*

Yahoo! Inc., 2011. Yahoo! Design Pattern Library. *Online Resource*. http://developer.yahoo.com/ypatterns/

Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N., 2009. Managing architectural decision models with dependency relations, integrity constraints, and production rules. *Journal of Systems and Software 82(8): 1249-1267.*