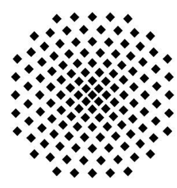


Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-based Application Architectures

Christoph Fehling[1], Frank Leymann[1], Ralph Mietzner[1], Walter Schupeck[2]

Report 2011/05
Mai 10, 2011

[1]Institute of Architecture of Application Systems
Universitätsstr. 38
70569 Stuttgart
Germany

[2]Daimler AG
Epplestraße 225
70546 Stuttgart
Germany

CR: C.0, C.2.4, D.2.2, D.2.3, D.2.7

Caveat

Describing good solutions to reoccurring problems faced during the development of software systems as patterns is a common practice in research and industry alike. While the development of cloud applications faces many new challenges, existing patterns can be applied directly or can be transferred to the area of cloud computing with adjustments. This catalog therefore contains existing patterns from the areas of standalone applications, grid applications, and message-based applications. These patterns where transferred to the area of cloud computing focusing specifically on this environment. They were compiled into the catalog using the same form as new patterns to increase readability and accessibility. We do not claim to have invented these patterns. References are given to the original sources in the respective sections of this document.

Document Versions

Version	Date	Change
1.0		Initial version of this document.

Index

Abstract 5
1 Introduction 5
1.1 Pattern-based Description for Cloud Computing 6
1.2 Document Structure 6
1.3 Visual Representations 6
2 Cloud Basics 8
2.1 Cloud Service Models 8
2.1.1 Infrastructure as a Service (IaaS) 9
2.1.2 Platform as a Service (PaaS) 10
2.1.3 Software as a Service (SaaS) 12
2.1.4 Composite as a Service (CaaS) 13
2.2 Cloud Types 15
2.2.1 Public Cloud 15
2.2.2 Private Cloud 17
2.2.3 Community Cloud 18
2.2.4 Hybrid Cloud 20
3 Cloud Service Types 22
3.1 Cloud Compute Services 22
3.1.1 Elastic Infrastructure 22
3.1.2 Low Availability Computing Node 24
3.1.3 High Availability Computing Node 25
3.2 Cloud Storage Services 26
3.2.1 Strict Consistency (Service Behavior) 26
3.2.2 Eventual Consistency (Service Behavior) 27
3.2.3 Relational Data Store 29
3.2.4 Blob Storage 30
3.2.5 Block Storage 31

3.2.6 NoSQL Storage 32

3.3 Communication Services 33

3.3.1 Message-Oriented Middleware 33

3.3.2 Reliable Messaging 35

3.3.3 Exactly-once delivery 36

3.3.4 At-least-once delivery 37

4 Cloud Application Architecture Patterns 39

4.1 Basic Architectural Patterns 39

4.1.1 Composite Application 39

4.1.2 Loose Coupling 41

4.1.3 Stateless Component 42

4.1.4 Idempotent Component 43

4.2 Elasticity Patterns 45

4.2.1 Map Reduce 45

4.2.2 Elastic Component 46

4.2.3 Elastic Load Balancer 48

4.2.4 Elastic Queue 49

4.3 Availability Patterns 51

4.3.1 Watchdog: High availability with unreliable Compute Nodes 51

4.3.2 Update Transition 52

4.4 Multi-Tenancy Patterns 54

4.4.1 Single Instance Component 54

4.4.2 Single Configurable Instance Component 55

4.4.3 Multiple Instance Component 56

5 Acknowledgements 58

6 References 58

Abstract

Patterns are a widely used concept in computer science to describe good solutions to reoccurring problems in an abstract form. Such conceptual solutions can then be applied in concrete use cases regardless of used technologies, such as software, middleware, or programming languages.
As cloud computing is a new and developing field of commerce, new products and technologies are constantly made available to cloud users. In this scope, market dynamics often lead to confusing service descriptions. While advertising the individual properties of a specific cloud service may help in positioning it on the competitive market of cloud computing, they obfuscate the common underlying concepts. In this report, we therefore employ a pattern-like description language to describe cloud service models and cloud types in an abstract form to categorize the offerings of cloud providers. Further, we give reoccurring architectural patterns on how to design, build, and manage applications that use these cloud services. The abstracted form of these architectural patterns make them applicable to challenges that developers of cloud application face today, independent of the actual technologies and cloud services that they are using.

1 Introduction

Cloud computing drastically changes the way how IT resources, such as servers, applications, and storage are accessed and used. Instead of providing these resources in dedicated and private data centers, companies access them on-demand over a network. Traditional APS providers or pure virtualization environments can be differentiated from clouds by three properties.

Elasticity – resources in a cloud can be reserved and freed flexibly, often within minutes. This allows the actual number of such resources to be aligned tightly to the current demands of a company using cloud technologies. Applications that experience a higher load during a certain time of the year, for example, can request more resources only during these periods. Especially, for development and test purposes, this property of clouds can make their use very profitable.

Pay-per-use – no monthly charge for resource use is applied. Costs only arise for resources during their usage times. Therefore, no long-term upfront investments (CAPEX) in IT resources are required anymore. Instead, only the operational costs (OPEX) of these resources arise.

Standardization – through the use of hardware virtualization and the resulting increase of image-based system management standardizes the used hardware software stacks that are used in cloud applications. Especially, differences between versatile hardware platforms are avoided, because a homogeneous virtual hardware platform is provided on top of them. Virtual servers, used in cloud computing, can therefore often migrate between multiple physical hardware platforms easily, as long as the same virtualization environment can be provided.

1.1 Pattern-based Descriptions of Cloud Computing

The evolution of cloud computing has been mainly industry-driven and remains at this stage today. Many new concepts and technologies are being developed rapidly and underlying concepts are often assumed implicitly. The introduction and use of cloud technologies in a company are often hindered by these circumstances. We therefore introduce an abstract description of cloud service models, cloud types, types of the offerings provided in these clouds, and architectural patterns, describing how applications are commonly built on top of clouds. Descriptions of these elements all show a pattern-based form. According to the pattern-based description used in this report, each pattern is identified by an icon and a question that motivate its use. Further, the context and challenges of the pattern are given to define the environment in which it is applicable. Then, a solution to the challenges is described with help of a sketch depicting the fundamental components of the pattern. The solution is very brief and only states in short the actions that are taken to apply a pattern in the context. Detailed results are covered afterwards, followed by a list of other patterns related to it. This list is the main cause, why a pattern-based form was also employed to describe the other aspects of cloud computing, such as service models. For example, patterns may be only applicable if a certain service model is used, thus, by using the same descriptive language these relations can be easily expressed. After the description of relations to other patterns, the used pattern language gives variations the current pattern. A variation is a small adjustment of the pattern or a minimal different use of it. It however is not significant enough to justify its descriptions as a new pattern. A pattern description is finalized by giving real world examples where this pattern has been applied as well as further references relevant to the pattern. To increase the readability of this report, additional to a complete list of references at the end of the document, references are also listed within each pattern description.

1.2 Document Structure

The further structure of this document is as follows: Section 0 describes the basics of cloud computing comprised of the service models that are employed by the different types of clouds. Section 0 describes the abstracted types of services that are offered by clouds and their behavior. Computing services, communications services, and storage services are differentiated. Section 4 then describes architectural patterns how applications can be built on top of these cloud service types.

1.3 Visual Representations

The icons for patterns as well as the sketches use several reoccurring graphical elements.

Icon	Description
	Compute Node – Resource that can be reserved by users of a cloud or that may reside in a traditional static data center. This resource can be considered as a (virtual) server on which applications can be installed.
	Cloud – Used to depict a cloud computing environment that is offered to different customers and which may provide versatile resources. Generally, it is assumed that a cloud environment displays three characteristics: elasticity, pay-per-use, and standardization.

Icon	Description
	Company – Entity that uses IT resources and usually has a large number of employees associated with it that access these resources on behalf of the company. This icon is used to depict the affiliation of computing environments to a company.
	Access Control – This icon represents a component that handles user authentication and authorization when they access a controlled environment.
	Platform – Services that are offered on top of compute nodes are referred to as a platform. Again, applications may be executed using such a platform but a larger portion of application functionality and management functionality is provided by the platform. In some cases the notion of (virtual) servers on which such a platforms is based can be completely obfuscated.
	Database – Component that stores different forms of data that can be queried and accessed by users. Capabilities of these queries and the type of accesses differ greatly depending on the type of the database.
	Data Elements – Entities that a user stores in a database. Different forms indicate different information that given by the data elements.
	Globe – Used to depict the affiliation of computing environments to a general user group. These environments can therefore be considered to be accessible by a large number of individuals or companies.
	File – Element stored in a directory structure. It can be identified by a unique name, which is also used to access it.
	Hard Drive – Entity that is accessed as a block storage device having a specific formatting. A hard drive can either be a physical drive or a file that is merely accessed similar to a physical hard drive.
	Database Tables – Tabular entries in a database that can be queried. A bar on the top of these table elements is used to depict that the elements are structured. The lack of such a bar is used to depict that this structure is not existent or only very weak.
	Message – A small amount of structures information exchanged by communication partners. Usually, this exchange is asynchronous.
	Message Cannel – Link between two or more communication partners by which information (for example messages) are exchanged.
	Application Component – Entity that comprises a componentized application. An application component offers a certain set of application functionality via a well-defined interface to be used by other application components.
	Configuration – Parameters that specify how an application component shall behave when accessed by a certain customer.
	Dashed Versions – Any element may also be depicted in a dashed form, which is used to express one of three conditions. (1) The element is added or removed. (2) The element is a virtual representation of its physical counterpart, that either provides access to the physical counterpart or contains information how to instantiates the physical counterpart.

2 Cloud Basics

In the following it is described, how IT services are offered using cloud computing as well as the different types of computing clouds and their specifics. The cloud service models and cloud types described here are conformant to the NIST definition of cloud computing [54].

2.1 Cloud Service Models

Cloud Service Models describe how different types of resources are offered as a service by the cloud provider. Depending on the portion of the application stack that is controlled by the provider, one differentiates between *Infrastructure, Platform, Software*, or *Composition as a Service* (IaaS, PaaS, SaaS, CaaS respectively). This correlation between the software stack and the different "as a Service" models is also depicted in Figure 1.

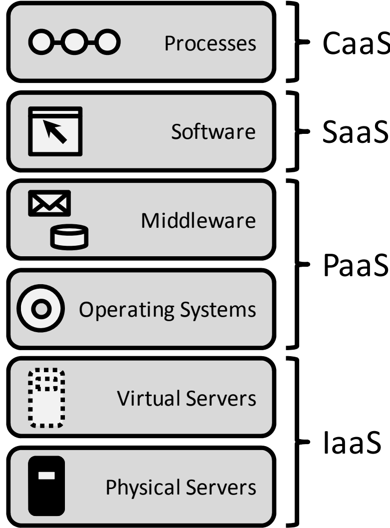


Figure 1: Correlation between Application Stack Layers and the different Cloud Service Models

Especially, the following cloud service models may rely on other resources also offered as a service. This is often the case, because the desired cloud properties (elasticity, pay-per-use, and standardization) of one service offering often require the same properties to be present in underlying application layers. The dependencies of cloud service offerings are depicted in Figure 2.

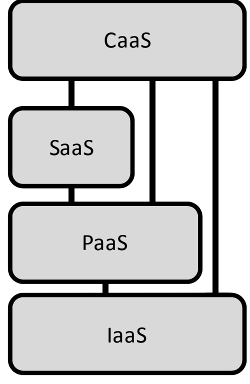
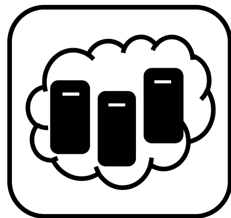


Figure 2: Possible Dependencies between Cloud Service Models

2.1.1 Infrastructure as a Service (IaaS)



How can IT-infrastructure be offered dynamically over a network?

Context

The (virtual) servers of an *elastic infrastructure* (3.1.1) shall be offered to different users that are isolated from each other on a pay-per-use basis.

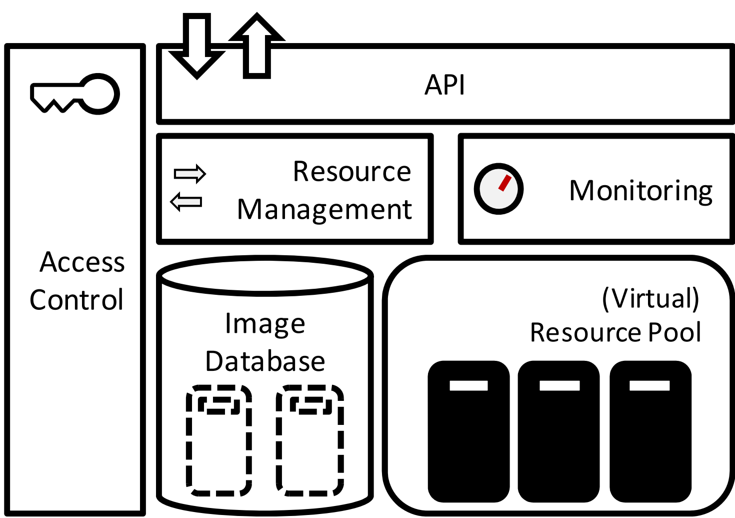
Challenges

Resources in an elastic infrastructure share common underlying resources such as networking, storage and optionally servers if virtual servers are provisioned on top of them. In such a setting different users of the infrastructure must be isolated regarding data and performance. It therefore has to be ensured that users cannot access other users' data and that resource utilization of one user does not affect the performance of another user. Also, additional support systems are needed to monitor the resource usage and bill them to customers accordingly.

Solution

Access control is added to an elastic infrastructure and the resource management is extended to isolate users from each other. The monitoring component collects additional information to support pay-per-use billing.

Sketch



Results

Access controls authenticates users and control their usage of the API during the management of (virtual) server images and the starting and stopping of (virtual) servers. Further, the monitoring component is extended to support billing based on accesses to the API as well as the amount of used resources. The resource management now ensures that one user does not utilize that many resources that it affects other users. Assured service levels are often expressed as a comparison to traditional systems, i.e. compute performance is said to be equivalent to a 1.6 GHz dual core CPU. Statements like this however have no indication how this performance is actually achieved.

Additional requirements regarding the isolation of running (virtual) server relies on technologies used in traditional server hosting environments, such as quotas for communication channel throughput.

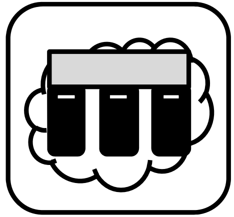
Relation to other Patterns
Higher functionality that constitutes the application stack, such as middleware, software, or individual composition of applications can also be offered as a service as described by the *Platform* (2.1.2), *Software* (2.1.3), and *Composition as a Service* (2.1.4) patterns respectively.

Infrastructure as a Service may be part of the offerings that form a *public* (2.2.1), *private* (2.2.2), or *hybrid cloud* (2.2.4). In this case it is ensured that resources are available at virtually infinite numbers.

Variations
Traditional dedicated server hosting could be compared to IaaS. However, it often lacks the dynamic pay-per-use billing model. Instead users pay monthly fees for servers. Similar billing models are also introduced to the IaaS market, so that users can decide between usage based pricing and a lower price if resources are reserved for longer time periods.

Examples / References
The first and still the most dominant provider of IaaS is Amazon EC2 [3]. Virtual images can be created based on pre-configured images or through extraction from running systems. The API used by Amazon to manage virtual machines has also been implemented by the open source Eucalyptus project [29]. Recently, Microsoft also introduced a VM Role [57] to its Windows Azure cloud platform as an IaaS offering. However, the operating systems that a user can install on this particular a virtual machine are limited to Windows versions, by the time of this writing.

2.1.2 Platform as a Service (PaaS)



How can IT platforms be offered dynamically over a network?

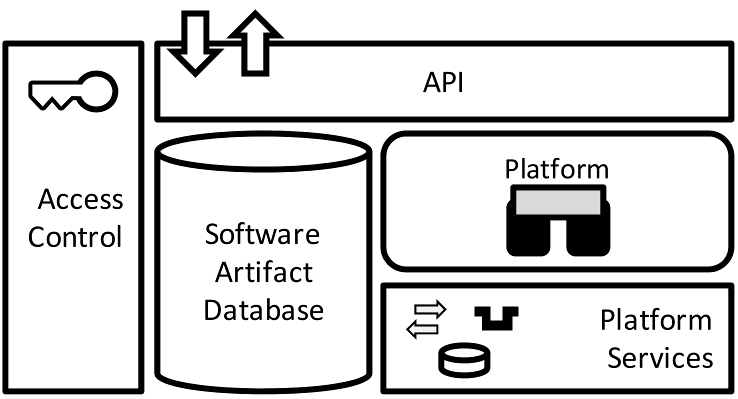
Context
A middleware platform shall be offered to different users that are isolated from each other on a pay-per-use basis.

Challenges
In order to host the software components created by multiple users in a shared runtime environment, that offers commonly used platform functionality, additional isolation of these components has to be insured. The platform itself therefore has to be made multi tenant-aware, so that software components cannot access data or functionality and do not influence the performance of other users' software components. Also, the platform needs to offer common services that can be used for communication between these components, data storage and the routing of accesses to components. Elasticity of the hosted components shall also be enabled automatically.

Solution

An API allows users to deploy software components to a *Platform as a Service* offering, register and configure other platform services for communication e.g., message queues, storage e.g., block storage, and routing e.g., realized in an enterprise service bus.

Sketch



Results

Accesses to deployed software components and registered platform services are controlled to ensure isolation of users. The middleware components, such as applications servers, enterprise service busses, and messaging systems are extended to assure equivalent performance to all users. Software components are often created using specific development environments or libraries to ensure certain component properties, such as statelessness for example, to enable a platform controlled elasticity of deployed applications. Billing services, also offered by the platform, are often based on the amount of storage used, number of messages sent, or accesses to the hosted services.

Relation to other Patterns

The offered platform services follow cloud infrastructure patterns. Communication between components can be realized using *messaging patterns* (3.3) and storage using *cloud storage patterns* (3.2).

Platform as a Service may be build upon an *Infrastructure as a Service* (2.1.1) to enable elasticity on the underlying hardware on which the offered middleware services are hosted.

To enable multi-tenancy of the offered middleware components, such as applications services, messaging systems etc., they implement *multi-tenancy patterns*. Platform as a Service may be part of the offerings that form a *public* (2.2.1), *private* (2.2.3), or *hybrid cloud* (2.2.4). In this case it is ensured that resources are available at virtually infinite numbers.

Variations

PaaS is often realized on top of IaaS by providing automated management of IaaS resources. This sometimes leads to blurred boundaries between IaaS and PaaS, for example in Windows Azure (see below).

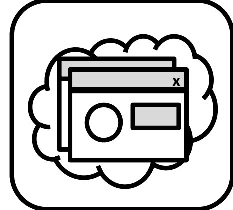
Examples / References

The Google App Engine [35] and Salesforce's Force.com [74] platform are pure PaaS offering where the user is unaware of the underlying infrastructure or its management. When using the Windows Azure [63] PaaS offering, he is still aware of virtual machines, so called roles, on which his custom

developed application components are hosted. These virtual machines are however managed by the platform regarding application update and patch management.

Such an approach is taken to allow users of the platform to configure the level of automatic management to their needs. Some application components can be hosted on platform managed virtual servers, while others are managed manually. This is especially useful when existing applications are moved to the cloud and it is unclear how they will react to automatic management processes.

2.1.3 Software as a Service (SaaS)



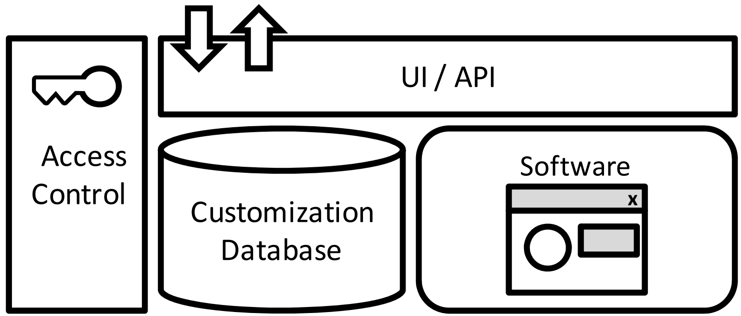
How can software be offered dynamically over a network?

Context
A software shall be offered to different users that are isolated from each other on a pay-per-use basis.

Challenges
In order to allow multiple users to access software over a network it has to be ensured that each user perceives the software as if he was the only user. Since it is often unfeasible to provide an individual instance of the software for every user, the software itself has to be made multi-tenant aware. This way, its components, such as the user interface, business logic, and data storage can be shared by multiple isolated users. Further, every user shall be enabled to configure the software to his needs regarding the location or content of menus in the user interface, for example, or the date and currency formats that shall be used.

Solution
A user interface or an API is used to access the *Software as a Service*. Access is controlled to ensure the isolation of multiple users while the desired customization is stored in a central database that controls how shared components behave.

Sketch



Results
The software is offered over a network using either a user interface or an API. In the former case, a user often accesses the software using a browser. In the latter case, the software provides

functionality that is integrated with a user's application that he runs on his own premise. For example, an external provider could offer communication services to send text messages or establish teleconferences. These services could then be integrated in a user developed calendar application.

Access to the hosted software is controlled to avoid that users can access other users' data or influence the performance that others experience. The customizations specified by users are stored in a database and are accessed from software components to determine their behavior.

Relation to other Patterns
Software as a Service may use underlying *Platform* (2.1.2) or *Infrastructure as a Service* (2.1.1) offerings to enable the desired elasticity. While this elasticity does not directly benefit the user of the Software as a Service offering, it allows the provider to scale the offered software dynamically with changing user demands. This is a very important aspect in the Software as a Service market, since the software offered often targets a large number of customers to leverage economies of scale.

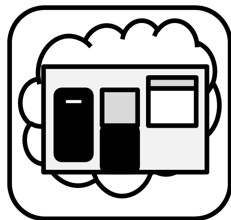
The components out of which the offered software is comprised can be implemented accordingly to *multi-tenancy patterns* (4.4) to ensure the required multi-tenant awareness.

Software as a Service may be part of the offerings that form a *public* (2.2.1), *private* (2.2.3), or *hybrid cloud* (2.2.4). In this case it is ensured that resources are available at virtually infinite numbers.

Variations
The concept of offering software over a network is known for a very long time as application service providers (ASP). However, in the past it has been realized mainly in a static fashion that hindered providers to scale dynamically. With the increased availability of cloud computing, the concept was revived and extended to exploit the elastic infrastructures that are now available.

Examples / References
The first significant provider of SaaS was Saleforce.com's web-based CRM software [73]. Microsoft offers its complete office and collaboration suite as a Service, as part of Office Live [56]. IBM also offers the collaboration software Lotus Live [42] as a service. Similar products, subsuming office and collaboration functionality, are available from Google, called Google Apps [36].

2.1.4 Composite as a Service (CaaS)



How can composite application be offered dynamically over a network?

Context
Different provider supplied services shall be offered to users that are isolated from each other on a pay-per-use basis. These users shall be enabled to create individual compositions of the provider supplied services to meet their functional and service level requirements.

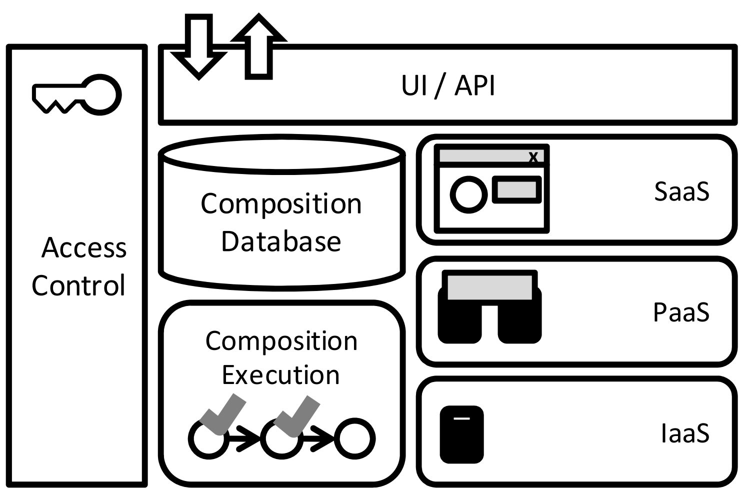
Challenges
Individualization of IT services regarding offered functionality and service levels often requires changes to the service implementation and the structure of the hosting environment. A good

alignment of user requirements and the offered services is however mandatory to increase the addressable market. The service provider therefore has to integrate functionally equivalent services assuring different service levels and run them in different hosting environments. However, creating all possible combinations of services to be selected from users is often unfeasible.

Solution

Users may create custom compositions of provider supplied services residing on the software, platform, or infrastructure layer. These customizations are again hosted by the provider.

Sketch



Results

The customer composes services offered by the provider to reflect the functional and service levels that he requires. This composition is uploaded to the provider and stored in a composition database. Access control ensures that the data and compositions of multiple users remain isolated. A user accesses a composition either through a user interface or an API depending on the type of service that is offered by the composition.

A runtime is included to execute the customer specified compositions. Different variations of this runtime could be available. For example, a CaaS provider could provide a BPEL or a Java Engine.

Relation to other Patterns

The services that are composed by users may be shared. In this case they should be implemented according to *multi-tenancy patterns* (4.4).

The composition execution engine may be provided as a *PaaS* (2.1.2) offering that supports the required composition languages.

Composition as a Service may be part of the offerings that form a *public* (2.2.1), *private* (2.2.3), or *hybrid cloud* (2.2.4). In this case it is ensured that resources are available at virtually infinite numbers.

Variations

As a special form of CaaS the provider can integrate *aaS offering of external providers. This may even include services that are created and maintained by the user of the CaaS offering. CaaS may

therefore form the basis for a *hybrid cloud* in which services of *public* and *private clouds* are integrated to allow users to combine the best of breed for their specific needs.

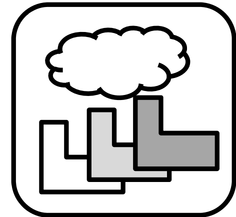
Examples / References
CaaS is still ongoing research. A good introduction and its relation to IaaS, PaaS, and SaaS are given in [53] and [21]. [71] describes its importance during the adaptation of software and its reconfiguration. Also, applications often the architecture of application components need to respect their composition as part of a CaaS offering, as described by [31].

Today, CaaS is already offered as online platforms that allow modeling and execution of business processes, such as RunMyProcess [72], Cordys Process Factory [25], or Intalio's online modeling tool [49]. Microsoft's process modeling capabilities, named Sharepoint Designer [58], also form a CaaS offering.

2.2 Cloud Types

Different types of clouds exist that can mainly be differentiated by the institution they are associated with, thus, the institution that is responsible for the operation of the cloud, and the targeted user group.

2.2.1 Public Cloud



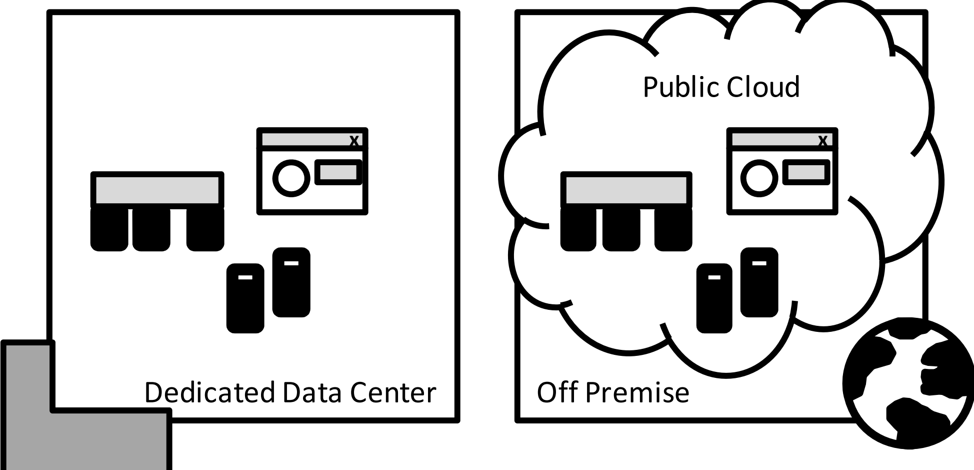
How can elastic IT services be offered concurrently to different companies?

Context
A single or combination of *aaS offerings shall be shared between multiple companies to reduce costs and enable a dynamic resource usage following a pay-per-use pricing model.

Challenges
The provider needs to maintain physical data centers that are limited in capacity but need to support the desired dynamicity and elasticity. Additional challenges arise for the provider and the customer, since in a public environment the users of the cloud generally do not trust each other. Therefore, providers and application developers alike need to address this increased security risk that is coming from inside the shared cloud.

Solution
The provider leverages economies of scale of the *public cloud* to allow customer to use resources dynamically while leveling the utilization of his static physical data centers. Additional security mechanisms are implemented to isolate customers from each other.

Sketch



Results

Due to the number of users as well as their geographic distribution, the utilization of the cloud is leveled. The peak-loads of customers can be handled since other customers require fewer resources during those times. Thus, the size of a public cloud enables the provider to provide dynamic and elastic resource usage, while ensuring a good utilization of his static physical data center.

Also, the traffic created by cloud customers is analyzed by the provider to detect unlawful use of cloud resources. In an IaaS environment, the additionally required security means to isolate (virtual) servers can be realized using established technology, such as firewalls. In a *PaaS* (2.1.2) and *SaaS* (2.1.3) environment the security isolation of tenants has to be addressed by the provider through adjustment of the offered middleware or software itself.

Relation to other Patterns

The public cloud can offer *infrastructure* (2.1.1), *platform* (2.1.2), *software* (2.1.3), *or composition as a service* (2.1.4) as well as a combination thereof.

Often, a public cloud provider offers the creation of private partitions and their integration to form a *hybrid cloud*.

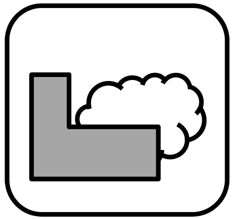
Variations

Trust is the major issue for the acceptance of public clouds. The fact that resources of different customers are hosted on the same infrastructure results in fears that malicious users of the same cloud could succeed in gaining illegal access to resources of other customers. Due to this fact, special public clouds are likely to be offered only to certain customer groups that in general trust each other. For example, a public cloud could only be accessible to customers from health care institutions. These clouds are a form of *community clouds*.

Examples / References

Amazon started to offer resources as a public cloud, called Amazon Web Services [2]. Recently a marketplace has been formed, called SpotCloud [28], where everyone can offer unused capacities in a similar fashion. The services of Windows Azure [63] and Salesforce's Force.com [74] are also publicly available.

2.2.2 Private Cloud



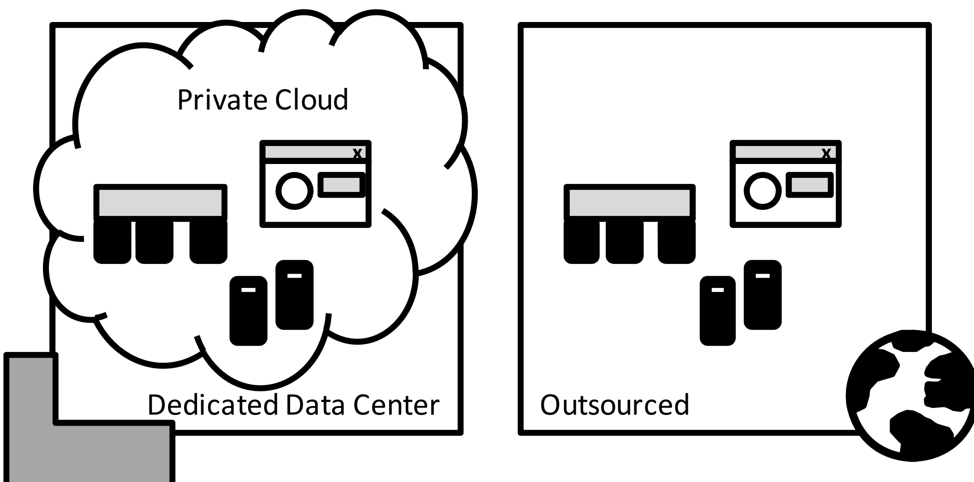
How can elastic IT services be offered exclusively for internal use of one company?

Context
A single or combination of *aaS offerings shall be used internally of a company since privacy requirements or restrictions by law render the usage of public clouds impossible.

Challenges
The company has to maintain physical data centers that are limited in capacity but shall provide a certain level of dynamicity and elasticity of the *aaS offering. Since the user group is however smaller than in a public cloud and often displays a regular usage pattern, economies of scale are harder to leverage.

Solution
A *private cloud* is established in a dedicated data center by the company itself or by an external provider. The services offered by this cloud are only accessible by one company.

Sketch



Results
A private cloud ensures a maximum level of security and privacy since its services are only accessible by one company. This however results in a reduction of the dynamicity that can be offered and often avoids using a pay-per-use pricing model. Also, the cloud itself cannot be scaled optimistically but has to handle peak usages of the company of worst case scenarios. These peak usages cannot be addressed with resources that other companies using a *public cloud* (2.2.1) do not need at that specific time.

Due to these facts, the introduction of private clouds within a company often causes a centralization of IT services to leverage economies of scale as good as possible. But even if a private cloud does not exceed the critical size, application development and maintenance can benefit from the enforced standardization and resource sharing.

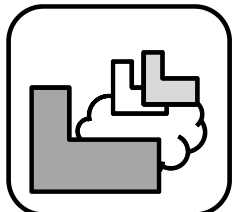
Relation to other Patterns
The private cloud can offer *infrastructure* (2.1.1), *platform* (2.1.2), *software* (2.1.3), *or composition as a service* (2.1.4) as well as a combination thereof.

Often, a certain part of the workload that is handled by a *private cloud* does not have high security and privacy requirements and could also be handled by *public clouds.* This portion of the workload is sometimes off-loaded to public resources during peak loads. This integration of private and public resources is called *hybrid cloud*.

Variations
A public cloud provider can offer a virtual private cloud that is a dedicated, specially secured partition of the public cloud. Network partitioning ensures that the virtual private cloud is isolated from other resources in the public cloud. Often, the virtual private cloud can additionally be integrated into the private cloud of a company using an encrypted communication link, such as a virtual private network (VPN) connection.

Examples / References
Tools that enable an elastic infrastructure, such as IBM Cloud Burst [39], VMware ESX [78], or Eucalyptus [29] can be used to create a private IaaS cloud. Amazon also offers the integration of public resources into a virtual private cloud [16].

2.2.3 Community Cloud



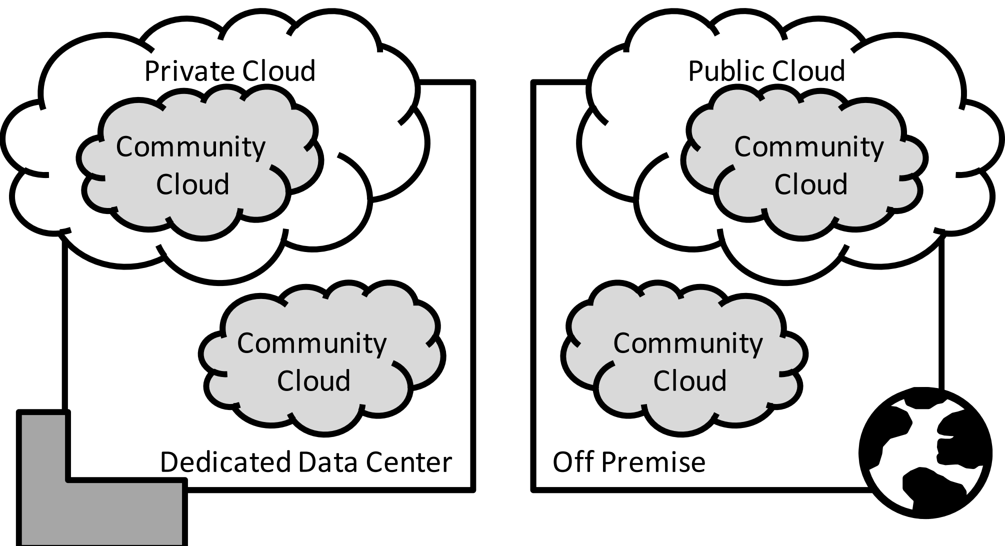
How can elastic IT services be offered concurrently to a certain set of companies?

Context
A single or combination of confidential *aaS offerings shall be shared between multiple companies that trust each other to reduce costs of the offerings and enable a dynamic resource usage and sharing.

Challenges
Often multiple companies must access shared resources to do business. For example, car manufacturers and their suppliers for physic simulations form such a scenario. Often parties trust each other since contracts have been established for the interaction. Therefore, computing resources must be made available to share confidential data. This generally renders the usage of public clouds impossible. Also, private clouds cannot be used since resources need to be accessed by multiple companies.

Solution
A *community cloud* is established containing computing resources that can be accessed by all business partners.

Sketch



Results

Depending on the concrete requirements on privacy, security, and trust, a *community cloud* can be an isolated portion of a *public cloud*, a *private cloud*, or a completely isolated computing environment. Isolation mechanisms are available in public and private clouds to avoid communication between certain resources residing in them. If the usage of a public cloud is inacceptable, the same isolation techniques can be established in a private cloud. However, if the maintainer of the private cloud does not trust the other companies enough to share his private cloud with them, a dedicated computing environment has to be established on or off premise for the community cloud.

Especially, a community cloud will contain services and data that all companies must use in conjunction for the specific business they are conducting.

Relation to other Patterns

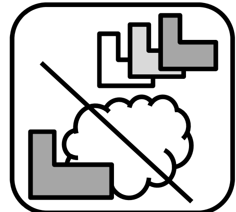
As said, a community cloud often is a special form of a *public* (2.2.1) or *private cloud* (2.2.2).

Examples / References

Generally, the same technologies as those used to create *private clouds* can be used to create community clouds that are managed in a dedicated data center. The only difference is that external companies are granted access. Virtual private clouds, like Amazon's VPC [16], do not natively support the access from different external companies. In this case one company would also be required to integrate the virtual private resources into its data center and then handle accesses from external companies.

Recently, Google started to provide its Google Apps in a community cloud for U.S. government agencies [37].

2.2.4 Hybrid Cloud



How can elastic IT services be offered to multiple companies, whilst some services are used exclusively by one company and may even be provided by it?

Context

A single or combination of the *aaS offerings shall be offered exclusively to companies with high privacy and security requirements while others shall be shared between multiple companies to reduce costs of the offerings and enable a dynamic resource usage following a pay-per-use pricing model.

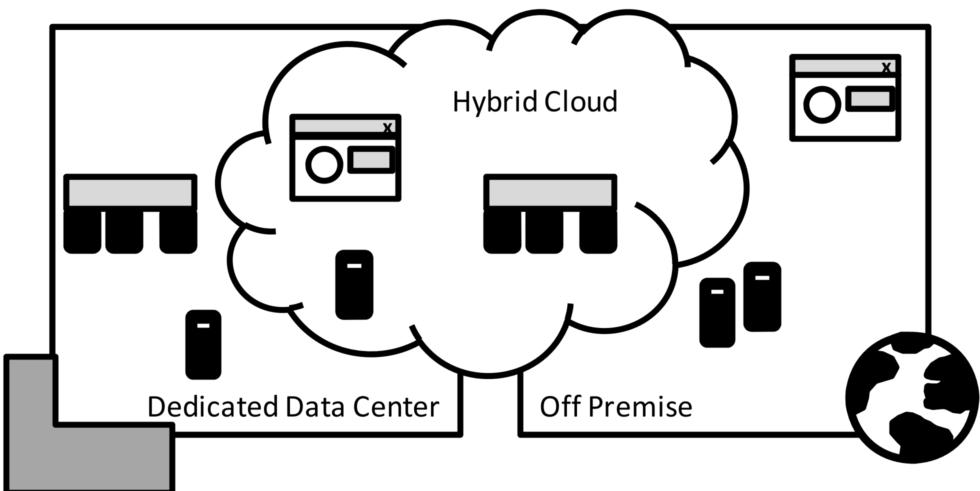
Challenges

Companies have different requirements on security, privacy, and trust, which determine whether they can share a cloud environment. Therefore, a company often has to maintain a private environment in which services with high security, privacy, and trust requirements are hosted. Other services without these high demands can be hosted in an additional public environment. Through an integration of these two environments better scaling and elastic behavior can be realized. However, this requires workload to migrate between environments.

Solution

Private and *public clouds* are integrated to form a *hybrid cloud*. Management functionality ensures that the disadvantages of private clouds (less elasticity) and public cloud (less privacy, security, and trust) are reduced by migrating workload between the environments.

Sketch



Results

If a company established a private cloud, it has to be scaled to handle peak-loads of the companies' workload. However, a certain portion of this workload and the services used often have lower requirements regarding privacy, security, and trust. This would make them a candidate for a public cloud. By integrating the private cloud with a public one and allowing services to migrate, this workload portion can be handled in the private cloud if utilization is low and by a public cloud during

peak-loads. If such a spill-over of workload is done, the private part of the cloud can be scaled more optimistically resulting in a higher utilization and a reduced operation cost of the overall environment.

Relation to other Patterns

The *hybrid cloud* can offer *infrastructure*, *platform*, *software, or composition as a service* as well as a combination thereof.

A hybrid cloud integrates services of *public* and *private clouds* and establishes common management functionality.

Variations

The private part of the hybrid cloud does not have to be realized using cloud computing technologies, such as virtualization, it can also be formed by a static data center.

Examples

The challenges to integrate resources residing in different clouds have to be addressed to form a hybrid cloud. This can be done using an enterprise service bus (ESB) that offers accesses to different resources in a seamless fashion. On-premise ESBs, such as Apache ServiceMix [19] or IBM WebSphere Enterprise Service Bus [46] can be used. Additionally, the ESB itself can be accessed as a Service from a cloud provider. For example, this is offered by Microsoft AppFabric [61], part of Windows Azure [63].

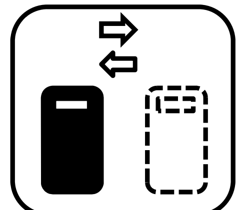
3 Cloud Service Types

A cloud offers different services following the different cloud service models described in Section 2.1. While the cloud service models describe the style in which resources are offered by a cloud, the cloud services types discussed in this section describe the nature and behavior of these services.

It can be differentiated between three cloud service types that are made accessible to a cloud user. *Cloud compute services* are used to execute workload. The required functionality may be part of the cloud offering or may be managed by the user through installation of custom software. Therefore, compute resources can be offered according to different cloud service models. *Cloud storage services* can be used to store data in the cloud. Cloud storage services differ greatly regarding the assured service levels and the displayed consistency behavior. *Cloud communication services* can be used to exchange information between the different applications and their components hosted in a cloud as well as to exchange information with resources residing outside of the cloud, for example, in traditional data centers.

3.1 Cloud Compute Services

3.1.1 Elastic Infrastructure



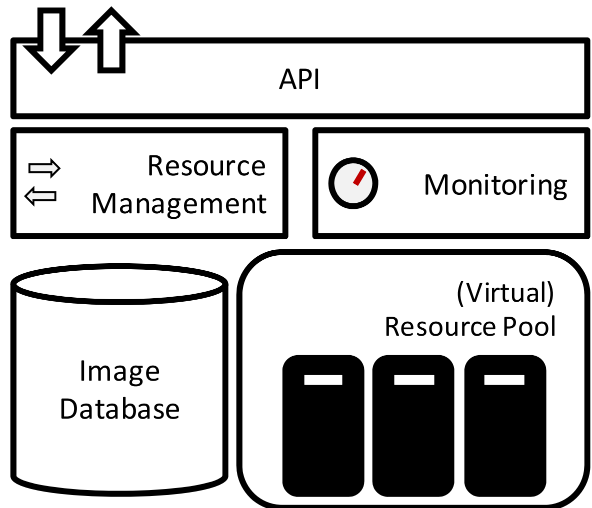
How can IT resources by offered dynamically and on-demand?

Context
Resources shall be assigned to and revoked from applications dynamically depending on the current load.

Challenges
The infrastructure must support dynamic provisioning and deprovisioning of resources. This requires the possibility to start and stop preconfigured (virtual) servers and their automatic integration in communication networks. Further, the infrastructure needs to allow the dynamic allocation of memory, processors, and storage etc. as well as the monitoring of these system resources to determine the utilization of the (virtual) servers. This functionality must be offered through an API to be used by atomized management tools and the applications that are hosted by the environment.

Solution
An *elastic infrastructure* allows the management of (virtual) servers and other resources, such as storage) through and API that offers start, stop, and allocation operations as well as monitoring of resources.

Sketch



Results

An elastic infrastructure supports the dynamic allocation of (virtual) resources that constitute a common resource pool. In case of server resources it allows dynamic starting and stopping of preconfigured (virtual) servers. This is enabled by storing so called server images in an image database. These (virtual) server images contain a description of the hardware configuration, the operation system, and possible additional middleware and software components.

Additionally, an elastic infrastructure contains a resource management component that handles the allocation of physical resources when requests are initiated via the API. Through a monitoring component information about (virtual) resources, such as utilization, may be extracted from the outside.

Relation to other Patterns

The elastic infrastructure forms the fundamental basis for cloud computing. It is therefore mandatory for the cloud service models, such as *Infrastructure as a Service* (2.1.1), *Platform as a Service* (2.1.2), or *Software as a Service* (2.1.3).

Variations

Typically, elastic infrastructures are built upon virtualized environments. However, there are tools that allow the elastic management of physical resource without an additional virtualization layer (see below).

Examples / References

VMware ESX [78], Xen [24], HyperV [55] etc. are classical virtualization environments that offer the functionality of an elastic infrastructure. Provisioning tools, such as IBM TSAM [45], IBM Tivoli Provisioning Manager [44], IBM Cloud Burst [39], OpenQRM [67] etc. provide fundamental functionality, for example server image management and the provisioning of complex system settings consisting of multiple virtualized servers that need to be configured coordinately. Also, these tools often handle the installation of operating systems and additional software on the managed servers.

3.1.2 Low Availability Computing Node

How can compute services be offered at low costs if their availability is relaxed?

Context

A computing environment shall be provided for services whose availability is not critical.

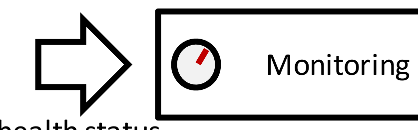
Challenges

The cost of commodity hardware has been reduced during the past years while its performance increased drastically. When combined in large numbers, these unreliable often virtualized servers can replace high performance solutions. However, the status of the overall system cannot be determined that easily anymore.

Solution

A large number of commodity (virtual) servers is used that provide the required performance and that provide the extraction of monitoring information.

Sketch



Results

The health status of (virtual) commodity servers is monitored so that a server failure can be detected.

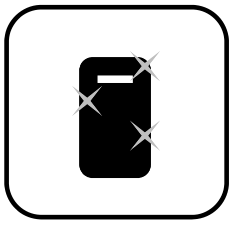
Relations to other Patterns

Low available compute nodes usually constitute a *public cloud* (2.2.1). Often, applications have higher availability requirements than what is guaranteed by low available compute nodes. The *watchdog* pattern (4.3.1) shows how an evaluation of the monitoring information and corrective actions can lead to a higher availability of the system composed of (virtual) commodity servers.

Examples

Many virtual servers of public clouds are offered at a low availability. Sometimes, availability is additionally expressed in an uncommon manner. For example, Amazon guarantees an availability of EC2 instances of 99.95% during a service year of 365 days [8]. However, this does not mean that a single instance has 99.95% availability during this time period, as could be expected. Instead, unavailability is defined as the state when all running instances cannot be reached longer than five minutes and no replacement instances can be provisioned. Additionally, the user has to make sure that instances are provisioned in multiple geographically distributed “availability zones”. Therefore, the availability of 99.95% can only be reached if multiple instances of an application are provisioned, and the *watchdog* pattern is implemented to ensure that new ones are started in case of failure.

3.1.3 High Availability Computing Node



How can compute services be offered if their availability is of vital importance?

Context

A computing environment shall be provided for services whose availability is critical.

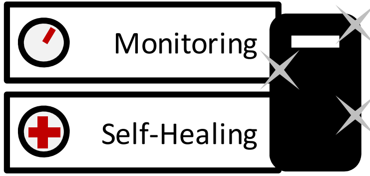
Challenges

Services important to a business have high demands on availability and performance. This has to be reflected by the environment on which they are hosted.

Solution

A *high availability compute node* is used that is specifically build to provide the required level of availability and performance through internal management functions.

Sketch



Results

The high available compute node continuously monitors itself and detects failures and performance problems. This information is used to react to faults and performance bottlenecks automatically. Often, this is enabled by redundant internal components that can replace failing ones.

Relations to other Patterns

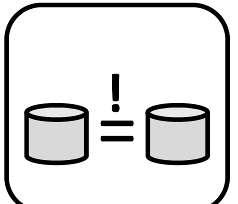
The availability provided by high availability compute nodes can also be realized with *low availability compute nodes* as described by watchdog: *high availability with unreliable compute nodes*.

Examples

Traditional mainframes such as IBM zSeries [43] are built to be fault tolerant and self-healing. They provide many virtualization technologies similar to those used to realize cloud computing. IBM Cloud Burst [39] and other technology used to build private clouds can also lead to compute nodes offering a high availability. Similar effects can be reached if a PaaS cloud is created on top of an IaaS offering and the middleware that is provided, such as IBM WebSphere [48], supports clustering.

3.2 Cloud Storage Services

3.2.1 Strict Consistency (Service Behavior)



How can the availability of a storage solution be increased while consistency is ensured at all times?

Context

A storage offering usually consists of multiple replicas to ensure fault tolerance. It is of major importance that the consistency of the data contained in these replicas is pertained at all times while the performance is of secondary importance.

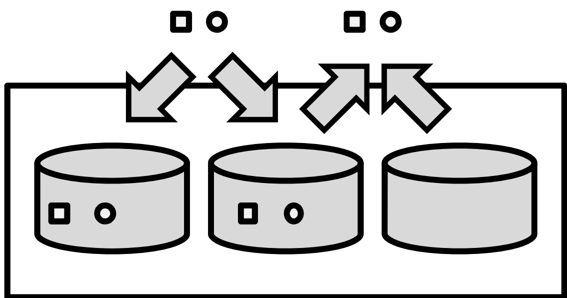
Challenges

The highest level of consistency is granted if all replicas are updated if the data contained by them is altered. However, this would mean that the availability of the overall storage solution is decreased drastically. It has to be ensured that it is available even if not all replicas are available, but still the correct version of the data is read.

Solution

Access only subsets of replicas during read and write operations to increase the availability. The size of these sets guarantee that at least on replica is read with the most frequent version at all times. The used read and write operations are subsumed in an ACID transaction.

Sketch



Results

Subsets of the available replicas are accessed during read and write operations. Thus, the system is available even if not all replicas are accessible. Strict consistency is guaranteed through the size of the subsets of replicas that are read or written. Considering the overall number of replicas (n), the number of replicas access during read (r), and those accessed during write (w), it is ensured that $n-w<r$ holds true for every read and write operation. Therefore, each read accesses at least one more replica than the previous write, ensuring that at least one replica is accessed with the most current version. The values for w and r are usually fixed at design time and reflect the different requirements on read and write performance. If write performance shall be increased, then the number of replicas accessed during a write is decreased and those during read increased, for example.

Relations to other Patterns

All cloud storage patterns can guarantee *strict consistency* or *eventual consistency* (3.2.2) in case consistency requirements can be relaxed in favor of performance and availability increase.

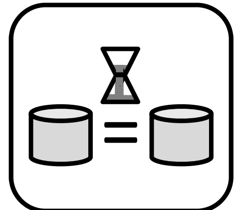
Variations

Some storage solutions allow the specification of consistency behavior on a per request basis. Therefore, critical information can be retrieved following strict consistency; less critical information is retrieved granting eventual consistency.

Examples / References

The ACID property of read and write operations is a transactional concept described in more detail in [20].

3.2.2 Eventual Consistency (Service Behavior)



How can the availability and performance of a distributed storage solution be increased if the requirement on consistency is loosened?

Context

A storage offering usually consists of multiple replicas to ensure fault tolerance. It is of major importance that the availability of the data contained in these replicas is increased while the consistency of the data is of secondary importance.

Challenges

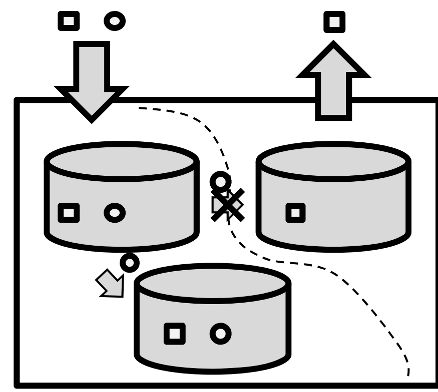
Assuring consistency among many (geographically distributed) replicas reduces the availability of the storage offering since it makes it dependant on the network over which replicas are updated. To handle possible partitions in this network, traditional replica models read and write a certain number of replicas to guarantee that at least one replica is read with the current version. Depending on the priority of read and write accesses the ratio of replicas that have to read or written can be adjusted. Therefore, either during read, write, or both multiple replicas must be accessible which reduces the availability of the overall storage offering. In certain scenarios consistency of data can however be relaxed to reduce the number of replicas to be accessed by operations. This increases the availability of the storage offerings since it is more robust regarding network partitioning.

It has however to be assured that changes to replicas are eventually propagated to all replicas. If some replicas are unavailable during a write the changes need to be stored at a different persistent location and need to be executed once the replica is accessible again. Additional challenges arise when replicas in different network partitions are changed independently and have to be merged once the network is not partitioned anymore.

Solution

Eventually consistent data storage allows reducing data consistency to increase availability and performance, since the impact of network partitioning is reduced and fewer replicas have to be accessed during read and write operation.

Sketch



Results

Eventually consistent databases increase the availability during network partitioning at the expense that inconsistent data can be read under certain conditions. This is achieved as follows:

While strictly consistent databases ensure that always at least one of the current version is read, eventually consistent databases allow that obsolete versions may also be read. This increases the availability of the storage offering since only one replica has to be available to successfully execute a read operation. Using this database model it can also be avoided that writing multiple replicas has to be executed as a distributed two-phase commit (2-PC) to guarantee ACID behavior. Instead replicas are updated asynchronously via transactional massages, for example. This guarantees that after a network partitioning problem is eventually resolved, changes to data are propagated to all replicas. A "consistency window" specifies how long this update via messages takes in absence of network partitioning. However, using eventually consistent storage also demands a certain data scheme and / or operation that are idempotent, so that changes to partitioned replicas can eventually be merged.

Relations to other Patterns

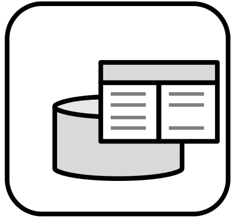
To exchange update information among (distributed) replicas, *reliable messaging* (3.3.2) can be used. To increase the performance and availability of this messaging system itself, storage of the messaging can again be realized using eventual consistency.

To ease merging of updates, the replicas should be implemented as *idempotent components* (4.1.4).

Examples / References

Amazon SimpleDB [15] uses two consistency models, strict consistency and eventual consistency. For every request to the storage offering a user can specify the required consistency model. In case of eventual consistency fewer replicas are read to increase the availability and performance. In case strict consistency is required, the number of replicas that are read is increased to guarantee accessing the current version. [79] gives an overview of the relation between consistency, performance, and availability. This relationship is also called the CAP (consistency, availability, performance) theorem stating, that in a distributed storage environment, only two of those properties can be optimized. It has been initially introduced by a PODOC keynote [22]. A more formal specification can be found in [34].

3.2.3 Relational Data Store



How can data elements be stored so that relations between their attributes and those of other elements can be expressed and complex queries based on these attributes are possible?

Context

An application uses a central database for storing data elements and performs complex queries on them.

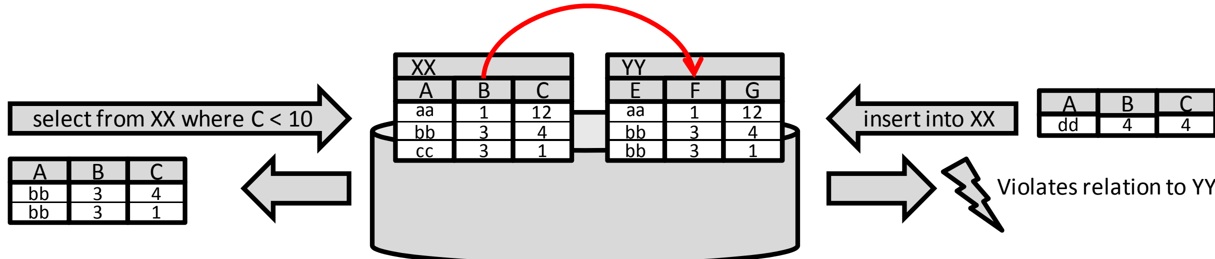
Challenges

Applications often access a database remotely and perform complex operations on the contained data elements. Queries are sent to the database to retrieve matching data elements. The more precise these queries can describe the required elements, the less stress is put on the network transporting data elements and on the application processing them.

Solution

Describe relationships between data elements and verify consistency of these relationships during data element modification. Allow complex queries on the data elements.

Sketch



Results

The relationships between attributes of elements in one table and elements in another table are expressed in a database schema that describes the structure of the database tables. Whenever a data element is created, altered, or deleted it is verified that the relations described in this schema are fulfilled.

Complex queries can be send to the database that express ranges of and conditions on data element attributes. Only the attributes that match the specified conditions are returned to the querying application. SQL is a common query language for this purpose (see below).

Relations to other Patterns

If the expression of complex queries is not needed scalability and performance of the data base can often be realized at lower costs, as described by *NoSQL storage* (3.2.6).

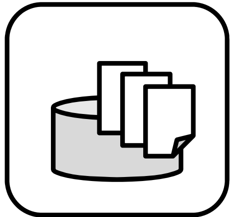
Relational data stores can display *strict consistency* (3.2.1).

Examples / References

A relational data store is offered by traditional data base systems, such as IBM DB2 [40], Oracle RDBMS [69], MySQL [68], or Microsoft SQL Server [60]. These can be also realized on top of an IaaS

cloud. Virtual machines offering such functionality are already available in Amazon EC2 [7], [6], [11]. Alternatively, PaaS offerings, such as Amazon Relational Database Service [12] or Microsoft SLQ Azure [59], can be used directly.

3.2.4 Blob Storage



How can large data elements be stored and organized centrally and made available over a network?

Context

A distributed application needs to manage large data elements, such as virtual server images or videos, which are too large for traditional databases.

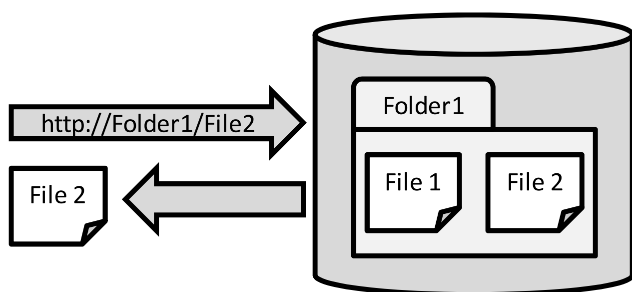
Challenges

In a distributed application data elements must be made available to all application components and to distributed users. Access to the data needs to be performed in a standardized fashion and access control has to be established.

Solution

Organize the data elements in a folder hierarchy similar to a traditional file system. Give each data element a unique identifier that can be used to access it over a network. Also, establish access control mechanisms.

Sketch



Results

The data elements are stored centrally and in hierarchical folders. Within each folder every data element is given a unique name. The combination of the position of an element within the folder structure, its unique name, and the service address form the address at which the element can be accessed. This access is enabled using established technologies, such as FTP, Rest over HTTP, SOAP over HTTP etc. (see references).

Relations to other Patterns

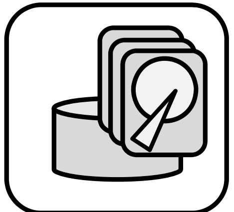
The data storage used for virtual images or software artifacts used by *Infrastructure* (2.1.1) and *Platform as a Service* (2.1.2) is usually realized as a Blob Storage.

Variations
The number of hierarchy levels in the directory structure is sometimes reduced (see S3 below). Also, sometimes automatic distribution about multiple geologically distributed replicas is offered to guarantee locality of data (see CloudFront below).

Examples / References
Traditional Web an FTP servers function in a very similar fashion. Amazon's S3 [14] services offers similar functionality but only allows folders without subfolders (called buckets). Especially, when accessed via a browser an index.html is not provided as usual for Web servers. Amazon CloudFront [5] is another service that provides similar access to streaming content that is automatically replicated geographically to increase performance.

In Windows Azure similar functionality is provided by Windows Azure Storage [62], a service that subsumes a *message oriented middleware* (3.3.1), *NoSQL storage* (3.2.6), and *block storage* (3.2.5).

3.2.5 Block Storage



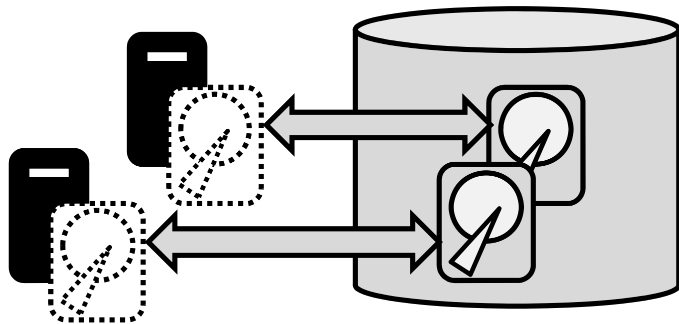
How can central storage be accessed similar to local hard drives?

Context
Servers forming a distributed system shall access a central high available storage as if it was a local drive.

Challenges
Resources in clouds are often unreliable (see *low available compute node 3.1.2*). Therefore, the data that they access locally shall in fact be stored in a high available central data store. This way, if a server fails the data is not lost, but a new server can be started to use the secured data.

Solution
Offer data elements in a central storage that can be accessed by distributed servers and integrate them as local drives.

Sketch



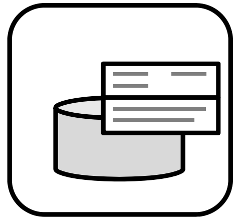
Results
Server can integrate centralized files into their system and treat them as if they were regular hard drives. Thus, they are formatted with a certain file system and accessed through the operating

system of the server. Often, these servers are virtualized, since the integration of files as hard drives is a basic functionality of virtualization software.

Relations to other Patterns
Often, *block storage* is used in conjunction with *low available compute nodes* (3.1.2) to allow recovering in case the node fails. Also, *blob storage* (3.2.4) is often integrated to create snapshots of block storage data elements.

Examples / References
Block storage is offered as a component of the Windows Azure Storage Service [62] and Amazon EBS [9].

3.2.6 NoSQL Storage



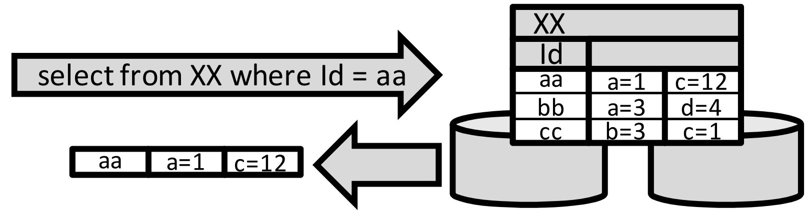
How can a database support extreme scale-out and a flexible data structure?

Context
Data storage shall be provided that is distributed among many resources and that frequently has to handle data structure alteration.

Challenges
Traditional relational database management systems are often hard to scale-out, for example due to dependencies between tables arising from foreign keys. The complexity of join operations, if data from remote systems has to be combined, forms an additional challenge. Those database systems are usually configured to utilize the resources of a central server or cluster optimally. However, cloud based applications usually do not have access to centralized, high performance servers but instead to a large number of distributed, commodity systems. These applications need to handle very large amounts of data and also need to be adjusted to new user demands flexibly. Therefore, a database solution is required that focuses on scaling out rather than on optimizing the use of a single resource and that can adjust flexibly to changes of the data structure.

Solution
Use a schema-free storage solution, with limited query capabilities to enable extreme scale-out through easy data replication.

Sketch



Results

The resulting NoSQL databases either do not support any schema at all or a very limited one. Often, they only allow querying a single index parameter. This allows them to be distributed among very many resources and the structure of the handled data remains extremely flexible.

This of course adds additional complexity to the application that uses such a data base. Changes to the implicit structure of the data need to be handled by the application as well as consistency checking. Also, no join operations are supported, which leads to generally more data returned to the application initiating the query. Traditional databases tried to reduce this amount of data through sophisticated join operations and conditional expressions. When using a NoSQL database this has to be implemented on the application level.

Relations to other Patterns

The additional load that is put on the application, due to weak query capabilities, is usually handled with a scale-out of the application. This can be realized by implementation of the *map reduce* (4.2.1) pattern.

To increase availability and performance even further, NoSQL storage solution often display *eventual consistency* (3.2.2). But *strict-consistency* (3.2.1) solutions are also possible. In the case of Amazon's SimpleDB (see below) this can even be specified on a per-request basis.

Variations

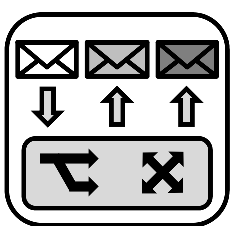
Since NoSQL is a quite new concept versatile implementations exists. Some allow only one data element to be associated with a key resulting in basically a distributed hash map implementation, others allow more data fields and additional structuring.

Examples / References

There are many products that offer NoSQL storage, such as Apache CouchDB [18]. Alternatively, it can be accessed as a service, for example Amazon's SimpleDB [15] or Windows Azure Storage Service [62]. The concepts of NoSQL storage are covered in detail in [30]. Google's Big Table NoSQL implementation is described in [23].

3.3 Cloud Communication Services

3.3.1 Message-Oriented Middleware



How can applications (or application components) communicate remotely via messages while being loosely coupled regarding their location and message format?

Context

Distributed applications or their components exchange information using messaging.

Challenges

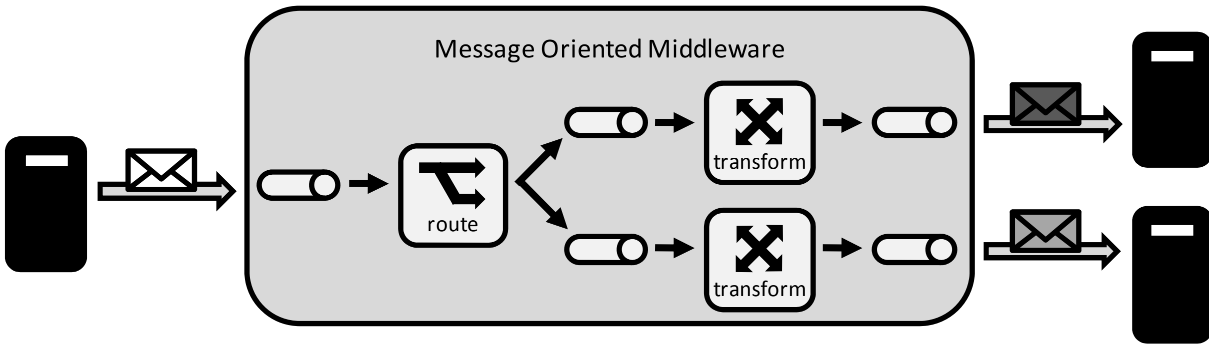
Different applications usually use different languages, data formats, and technology platforms. When one application (component) needs to exchange information with another one, the format of the

target application has to be respected. Sending messages directly to the target application results in a tight coupling of sender and receiver since format changes directly affect both implementations. Also, direct sending tightly couples the applications regarding the addresses by which they can be reached.

Solution

Connect applications through an intermediary, the message oriented middleware, that hides the complexity of addressing and availability of communication partners as well as supports transformation of different message formats.

Sketch



Results

Communication partners can now communicate via messages without the need to know the message format used by the communication partner or the address by which it can be reached. The message oriented middleware provides message channels (also referred to as queues). Messages can be written to these queues or read from them. Additionally, the message oriented middleware contains components that route messages between channels to intended receivers as well as handle message format transformation.

Relations to other Patterns

In a message oriented middleware messages are handled according to other patterns that influence the non-functional properties significantly. Those patterns are *reliable messaging* (3.3.2), *at-least-once delivery* (3.3.4), and *exactly-once-delivery* (3.3.3).

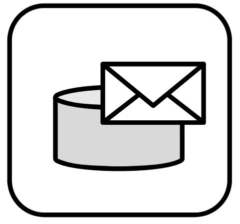
Variations

A message oriented middleware is often used, if data needs to be exchanged in a responsive manner. If larger amounts of data need to be exchanged applications can be integrated through file transfer or by sharing a common database.

Examples / References

Using messaging to integrate distributed applications is a common architectural approach. Many additional messaging patterns can be found in [38].

3.3.2 Reliable Messaging



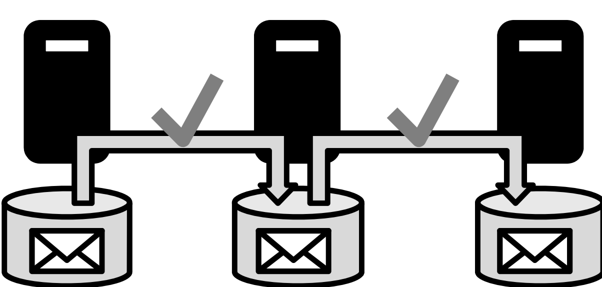
How can messages be exchanged while guaranteeing that messages are not lost in case of system or communication failures?

Context
Distributed applications or their components exchange information using messaging.

Challenges
When messages are exchanged in distributed systems, errors can occur during the transmission of messages over communication links or during the processing of messages in system components. Under these conditions it shall be guaranteed that no messages are lost and that messages can be eventually recovered after system failure.

Solution
Message exchange during communication partners is performed in under transactional context guaranteeing ACID behavior.

Sketch



Results
The message transfer from one communication partner to the other is performed under transactional context. Especially, this transaction subsumes the operation performed to store the messages in persistent storage. Thus, if an error occurs during message receiving, sending, or processing the transaction can be compensated transferring the overall system back to a correct and consistent state.

Relations to other Patterns
A *watchdog* (4.3.1) can be achieved if these nodes exchange information via reliable messaging.

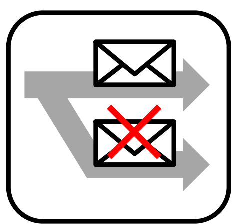
Since distributed transactions are difficult to implement and scale, sometimes the transactional behavior is relaxed. Instead of subsuming sending and receiving of messages in a transaction, the sender waits a certain amount of time for a receiver to acknowledge that the message was received. After this time frame the message is send again, which results in *at-least-once delivery* (3.3.4).

Variations
Sometimes not every communication partner has access to persistent storage. In this case the receiving of a message is contained in one transaction with its processing and the sending of another message to a communication partner that has access to persistent storage.

Examples / References

Reliable Messaging is supported by many messaging systems available today, such as Apache ActiveMQ [17], IBM Websphere MQ [47]. In the cloud, there are several messaging systems that can be accessed as a service, such as Amazon SQS [13] or the queue service part of Windows Azure Storage [62]. The cloud-based offerings however differ regarding the granted delivery model. They offer at-least-once delivery.

3.3.3 Exactly-once delivery



How can a message oriented middleware assure that a message send through it is delivered only once to a receiver?

Context

Communication partners exchange messages via a message oriented middleware.

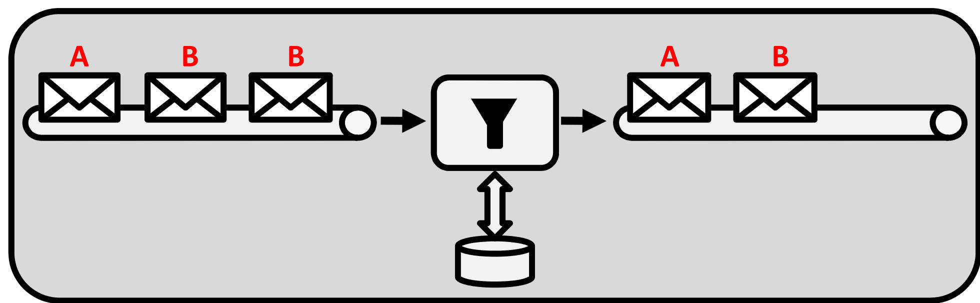
Challenges

Even though the use of *reliable messaging* (3.3.2) avoids duplication of messages in general, implementation specifics may still lead to message duplicates. This affected by the design decision how long to wait for a system to recover eventually from its persistent storage after it became unavailable. Sometimes the timeliness of messages demands resending them. Also, in some scenarios, especially regarding business to business interaction, *reliable messaging* may not be available at all. Under these conditions duplicate messages need to be handled.

Solution

Associate messages with unique identifiers and use filters to delete duplicates.

Sketch



Results

Whenever a message is created it is associated with a unique identifier. This is used by a filtering component on the message path to delete duplicates. It does so by storing the identifiers of messages it has already seen. The identifiers of messages passing through this filtering component are then compared to the identifiers that have been recorded to identify and delete duplicates. A central design decision is the size of the list that stores message identifiers, because it dramatically affects the robustness of the solution and its performance. Often, messages are associated with a time frame in which they are valid to limit the size of message identifier lists.

Relations to other Patterns

Exactly-once delivery is a non-functional property that some *message oriented middleware* (3.3.1) supports out-of-the-box. It drastically affects the way applications, relying on the messaging system, have to be built. Therefore, this has to be considered when identifying messaging systems suitable for certain applications.

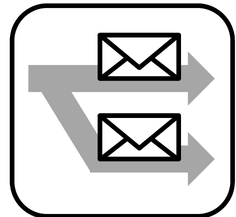
Variations

The filtering of messages can also be part of the receiver instead of being implemented in the message oriented middleware. This would then form an *idempotent components* (4.1.4).

Examples / References

The mentioned message filter is described as a separate pattern in [38]. Exactly-once delivery is extremely hard to guarantee. For example, IBM WebSphere MQ [47] can be configured to support it.

3.3.4 At-least-once delivery



How can the performance of a messaging system be increased if duplicate messages are acceptable?

Context

Communication partners exchange messages via a message oriented middleware.

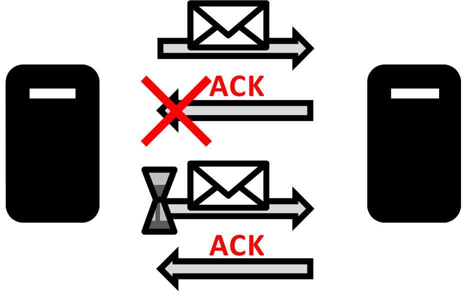
Challenges

Under some conditions receiving duplicate messages is uncritical. For example, if a database, like an organization employee directory, is queried using messages the re-execution of a query does not affect the state of the database. Therefore, the additional overhead to avoid the occurrence of message duplication during their transmission shall be reduced while still guaranteeing that a message is received.

Solution

Acknowledge that messages are received and retransmit the messages that have not been acknowledged.

Sketch



Results

The receiver of messages sends special acknowledge messages to the sender. If the sender does not receive such an acknowledgement message in a given time frame it retransmits the message. Thus,

messages, which are lost due to communication errors, are still received eventually. However, duplicate messages can occur, for example, if an acknowledgement message is lost.

Relations to other Patterns

Duplicate messages can either be handled by the messaging system itself which would then guarantee *exactly-once delivery* (3.3.3) or within an *idempotent components* (4.1.4).

Variations

To reduce the communication overhead, acknowledgement messages can be sent either after each individual message or after an agreed upon number of messages.

If sender and receiver of messages communicate via a queue the acknowledgment is not sent to the sender of the message but to the queue. A receiver removes the messages from the queue and acknowledges when he has finished processing it. Instead of deleting the message, after its removal from the queue, the messaging system keeps it in persistent storage. If the acknowledgement is not received after a certain time period, the message is put back on the queue.

Examples / References

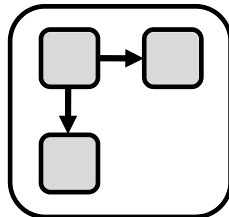
Today, all cloud messaging services guarantee the described at-least-once behavior. Amazon SQS [13] and Windows Azure Storage [62] service both implement the version of this pattern where messages are put back on queues automatically if receivers fail to acknowledge their processing.

4 Cloud Application Architecture Patterns

Now that cloud service model, cloud types, and the different types and the behavior of cloud services are defined, this section covers architectural patterns that describe how applications are build and managed using these cloud services. *Basic architectural patterns* are given that cover the fundamental structure of cloud application. *Elasticity patterns* specify how the size of a cloud application can be adjusted automatically to the currently experienced workload. These concepts are fundamental to ensure that the application can fully profit from the pay-per-use pricing models of clouds. It also ensures that economies of scale can be efficiently exploited, even in private clouds, because resources are shared best between multiple applications running in a cloud, if they are only reserved during the times they are needed. *Availability patterns* then cover how different levels of availability can be realized in cloud applications. This is especially important, because the availability of individual cloud resources is often significantly lower than that of resources in traditional data centers. Therefore, in the architecture of a cloud application the failure of individual resources often has to be anticipated to ensure an acceptable availability of the application. Finally, *multi-tenancy patterns* describe how cloud applications and their components can be shared by multiple users, so called tenants.

4.1 Basic Architectural Patterns

4.1.1 Composite Application



How can application functionality be distributed and composed from various sources?

Context

An application shall be composed out of multiple independent components.

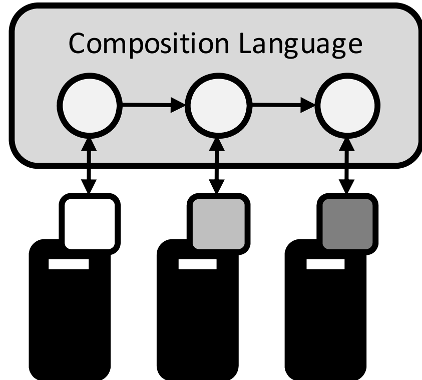
Challenges

Holistic applications are hard to integrate with other applications and inflexible when functionality shall be changed during the application lifecycle. In a cloud environment, application functions shall additionally be scaled-out individually and are often offered by different providers.

Solution

Divide the application functionality into multiple independent components that are integrated by the same means as if they were completely individual applications.

Sketch



Results

The application is divided into multiple components, each providing a certain set of functions. These components are integrated to form the functionality that the composed application shall offer. Due to this design the application is extendable right from the beginning and the integration of other applications is simplified. In fact, it blurs the boundaries between application components and complete applications by using the same integration techniques inside individual applications. A critical design decision is the distribution of functionality among components. If too few components are created, integration of new functionality and changing the application flexibly still remains a time consuming and error prone task. If the functionality is distributed among too many components, the resulting communication overhead is too high for the application to perform. In a cloud computing environment a fine grained distributed can be handled easier by scaling-out, a concept inherent to cloud infrastructures.

Often a special language is used for the composition of application components into a new application. A very important one in this context is the Business Process Execution Language (BPEL) that can be used to compose functionality offered by distributed Web services (see below).

Relations to other Patterns

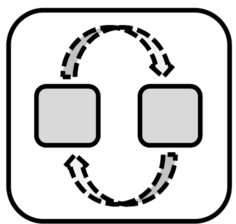
Usage of composite applications in conjunction with *loose coupling* (4.1.2) and a *message oriented middleware* (3.3.1) significantly increases the level to which an application can benefit from cloud properties, elasticity, pay-per-use, and standardized management.

Examples / References

BPEL [66] is a commonly used orchestration language for Web service offered via Interfaces that are specified in WSDL [80]. [52] gives a detailed background information about composition languages and BPEL. [81] does the same for Web services and resulting application architectures.

In [31], a framework is described how application components can be orchestrated in a user centric form. [77] generally motivates why applications should be split into separate components when using cloud computing.

4.1.2 Loose Coupling



How can the dependencies between applications and their components be reduced?

Context

A componentized application or a set of applications that shall be integrated.

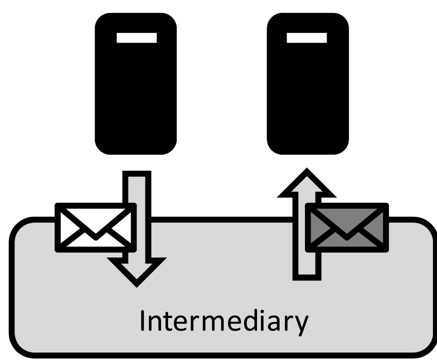
Challenges

In a componentized application, management processes, such as scaling, failure handling, or update management can be simplified significantly, because application components can be treated individually. This however demands that the dependencies among components are reduced, so that the addition, removal, failure, or update of one component has minimal or no impact on other components.

Solution

Decouple components by reducing the assumptions one component makes about another one when they exchange information.

Sketch



Results

The fewer assumptions two communication partners make about each other, the looser they are coupled and the more robust is the functionality they provide. At best, components communicate asynchronously through an intermediary that handles communication format transformation and message routing. Such functionality is usually implemented in a highly available middleware. Using this approach applications and their components do not know the concrete address of a communication partner, or the format used, and do not rely on the partner to be available at the time when the communication is initiated. This results in truly independent management processes of application components.

Loose coupling however comes at the price of performance reduction. Asynchronous communication via messages adds a lot of overhead to the information exchanged. Additionally, the communication path is longer since it includes address resolution and format transformation functionality. Therefore, when designing an application it generally has to be weighed between loose coupling and performance.

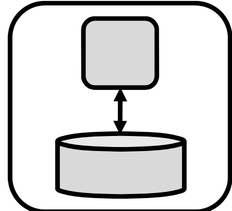
Relations to other Patterns

In an *elastic infrastructure* (3.1.1) the performance drawback of *loose coupling* is often less important and can be easily handled by scaling resources out. If *low availability compute nodes* (3.1.2) are offered in such a scenario, loose coupling is even necessary to avoid that the failure of one component affects others and thus it forms the basic concept behind the *watchdog pattern* (4.3.1).

Examples / References

Loose coupling is one of the fundamental concepts of service oriented computing (SOC). Its realization in a service-oriented architecture (SOA) is described in great detail by [81] and [50]. Loose coupling between applications and their components through messaging is described by [38].

4.1.3 Stateless Component



How can data loss be avoided if a component of an application fails or is removed from the application?

Context

A componentized application subsumes components that can fail.

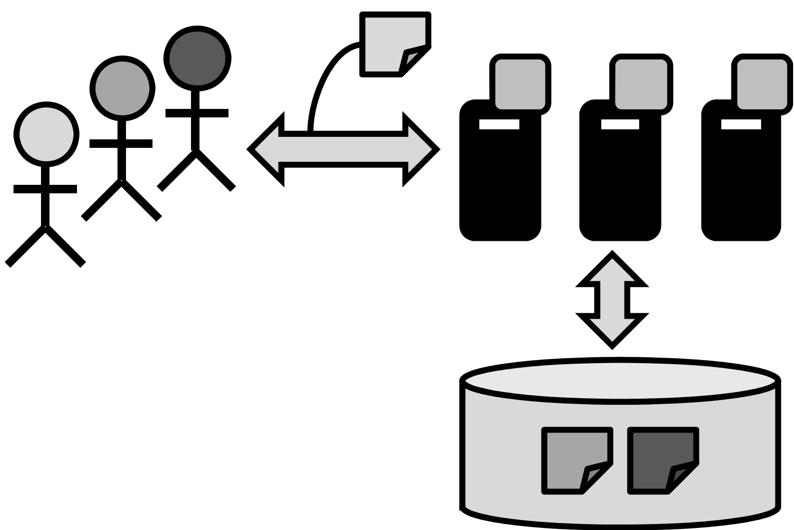
Challenges

If a componentized application is distributed among several compute nodes, the chance that a failure occurs is increased. This is especially the case if cloud resources are used, because those often display a very low availability. If the application shall be scaled out, component instances are also added and removed regularly when the demand changes.

Solution

Implement the components in a way that they do not contain any internal state, but completely rely on external persistent storage.

Sketch



Results

Since the component instances do not have an internal state, no data is lost if an instance fails. Such a setup also significantly increases the capability of the componentized application to scale-out, because multiple components can share a common data store and thus act as if they all had the same common internal state. Adding and removal of components, part of scaling operations, is also simplified. However, the scalability of the central data store becomes the major challenge when using stateless components.

Relations to other Patterns

To address the scalability challenge of the common data store, it is often distributed among several replicas that can be accessed by components. In this setup the consistency of the data store can also be relaxed, as described by the *eventual consistency* (3.2.2) pattern.

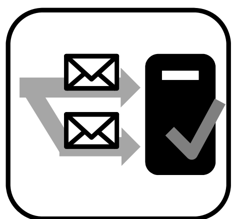
Variations

Instead of keeping the state in a common storage, it is sometime attached to each request that is send to the stateless component.

Examples / References

It is a central principle of the REST [33] architecture to transmit the state each time a component is accessed. Often, shopping baskets in web stores are realized this way. Web services can be developed with a similar approach [41].

4.1.4 Idempotent Component



How can a component receiving messages handle duplicate messages?

Context

Communication partners exchange messages via a message oriented middleware.

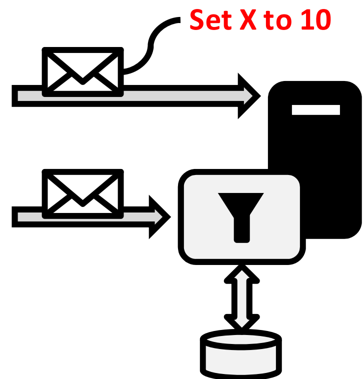
Challenges

If the message oriented middleware cannot guarantee exactly-once delivery, the receiver of a message must be able to cope with duplicate messages. Even if *reliable messaging* (3.3.2) is implemented and *exactly-once delivery* (3.3.3) is guaranteed, it may still be beneficial to implement receivers that can handle duplicate messages. This is especially the case if a packaged or legacy application has to be integrated that does not support transactions required by reliable messaging.

Solution

Implement the receiver in such a way that it can receive a message multiple times safely, either through a filter that removes already received messages or by adjustment of message semantics.

Sketch



Results

The desired behavior of receivers can be achieved by two approaches. First, a similar filter as the one used to ensure *exactly-once delivery* in the messaging system itself can be used to drop duplicate messages. Messages are associated with a unique identifier that is stored upon receiving a message. Based on the identifiers of subsequent messages it can then be ensured that duplicates are detected properly. Just as for the implementation of the filter within the message oriented middleware it has to be decided how long message identifiers shall be stored. However, a second alternative approach can be taken by designing the operations that are performed upon message receive to have the same effect when executed multiple times. For example, instead of sending a message saying that an account balance shall be increased by a certain amount, the message could specify the concrete amount to which the account has to be set. This can of course result in concurrency problems if messages are received out of order. In such cases message identifiers may also contain sequence numbers.

Relations to other Patterns

The idempotent component may implement a similar filter as the one used to ensure *exactly-once delivery* (3.3.3).

An idempotent component can be used to safely use a message oriented middleware that *guarantees at-least-once delivery* (3.3.4).

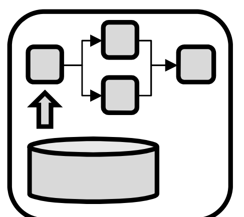
Examples / References

The mentioned two-phase commit protocol is covered in great detail by [20].

The idempotent component is also described as the idempotent receiver pattern in [38].

4.2 Elasticity Patterns

4.2.1 Map Reduce



How can the performance of complex queries on large data sets be increased if the used storage solution does not support such queries natively?

Context

A cloud based application that uses data storage that does not support complex queries, such as *NoSQL storage* (3.2.6).

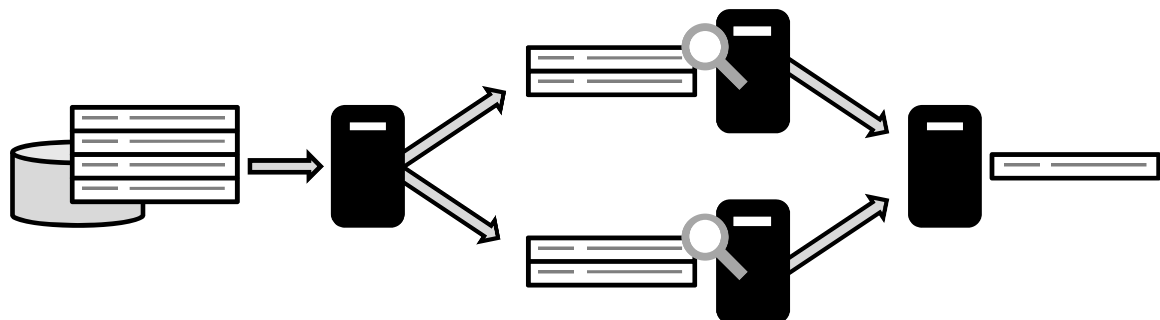
Challenges

For many cloud storage offerings, querying capabilities are limited to enable increased performance and scalability. This leads to larger data sets being returned to the applications that need to be handled efficiently. Since cloud applications need to be designed to scale-out, handling of complex queries has to be distributed among multiple compute nodes.

Solution

"Map" the data set into smaller sets and distribute it among multiple compute nodes, each executing the complex query on the assigned data set. Then "reduce" the solutions into one common data set that matches the complex query.

Sketch



Results

One compute node divides the querying data into smaller data chunks that it distributes among multiple compute nodes. Those perform the query on the data assigned to them and return a result. This result is again reduced to one consolidated query result by another compute node (this can be the same as the node dividing the data). During the reduction of results a simple merge can be performed, but more complex operations are also possible. Consider for example, that the query shall count the number of occurrences of a certain phrase in a large text file. This file is distributed among several compute nodes that count the occurrences in each part of the file and report it. These individual numbers are summed up by the reducing compute node.

Map reduce is used often to query large amounts of weakly structured data for analysis purposes. Examples are the analysis of web service logs to determine user access statistics or the analysis of order information to find popular products.

Relations to other Patterns

In messaging a similar pattern exists that distributes the handling of large, complex messages among several compute nodes, called *scatter-gather* (see below).

Often, *low availability compute nodes* (3.1.2) are used in conjunction with the *watchdog pattern* (4.3.1) since a large number of compute nodes are used by map reduces whose availability is often limited.

Variations

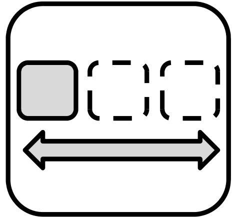
The map reduce pattern is not only used for data querying but can also be used to distribute other workload among compute nudes, such as physic simulations.

Examples / References

The mentioned scatter-gather pattern is described by [38].

The map reduce pattern is also described by [77] and [26].

4.2.2 Elastic Component



How can the number of application components, that are scaled-out, be adjusted automatically based on system utilization?

Context

A componentized application uses multiple compute nodes provided by an *elastic infrastructure* (3.1.1).

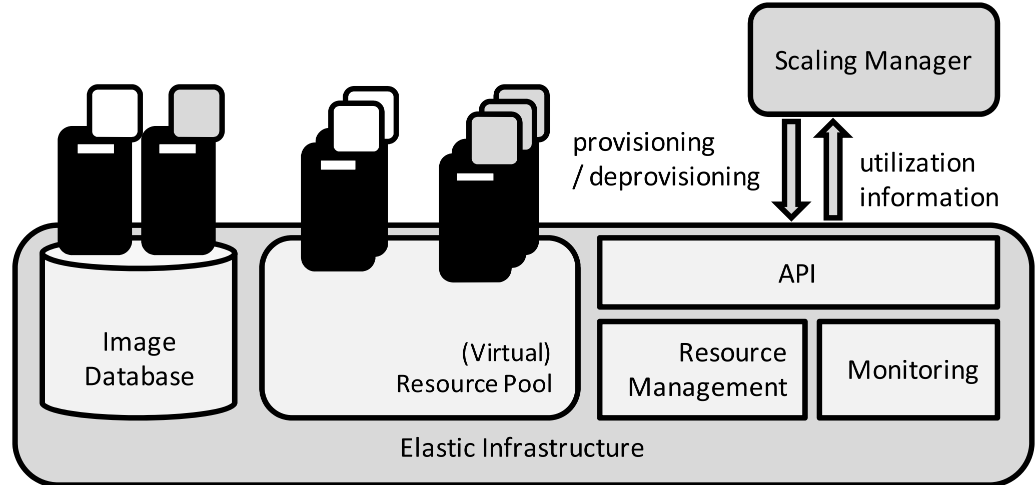
Challenges

In order to fully benefit from the dynamicity of an elastic infrastructure, the management process to scale-out a componentized application has to be automated. This way, the number of used resources can be aligned to changing workload quickly. If pay-per-use pricing models are available, this becomes even more significant, because the resource number directly affects the running cost of the application. Manual resource scaling would not respect this.

Solution

Monitor the utilization of compute nodes that host application components and automatically adjust their numbers using the provisioning functionality provided by the elastic infrastructure.

Sketch



Results

Since the application is componentized, its components can be distributed among multiple compute nodes. The system utilization displayed by these nodes, such as CPU load, memory usage, or network I/O, is monitored to deduct scaling decisions. If the utilization of compute nodes exceeds a specified threshold, additional hosting components are provisioned that contain the same application component. Often, several server images containing the application components are stored in the image management component of the elastic infrastructure to speed up this process. The distribution of application components among these images is a critical design decision that affects the granularity by which the application can be scaled-out.

Relations to other Patterns

To fully benefit from the monitored compute node properties, the application must use an *elastic infrastructure* (3.1.1).

In a *PaaS cloud* (2.2.1) utilization information of compute nodes can be unavailable or unsuitable to deduct scaling decisions, because compute nodes may be shared with other applications. In this case, scaling behavior must be realized on the application level either using an *elastic load balancer* (4.2.3) or an *elastic queue* (4.2.4). Even if utilization information of compute nodes is available and meaningful, the usage of such application level scaling techniques may allow the realization of a more sophisticated scaling behavior. First, they can monitor the utilization of individual application components independent on their actual distribution among compute nodes. Second, they can be used to actively influence the workload and adjust resource numbers environmental factors, such as fluctuating resource prices.

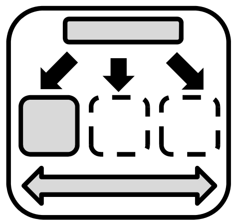
Variations

If the components are implemented as *stateless components* (4.1.3), the operations for adding and removal of components are significantly simplified. This is due to the fact that no internal state of the component has to be extracted prior to removal or inserted upon addition.

Examples

Many providers, such as RightScale [70] and ScalR [75], offer scaling functionality based on the utilization of virtual machines on top of Amazon EC2 [10]. Amazon also has its own service for this purpose, called Amazon Auto Scaling [4].

4.2.3 Elastic Load Balancer



How can the number of application components, that are scaled-out, be adjusted automatically based on the number of requests?

Context

A componentized application uses multiple compute nodes provided by an elastic infrastructure.

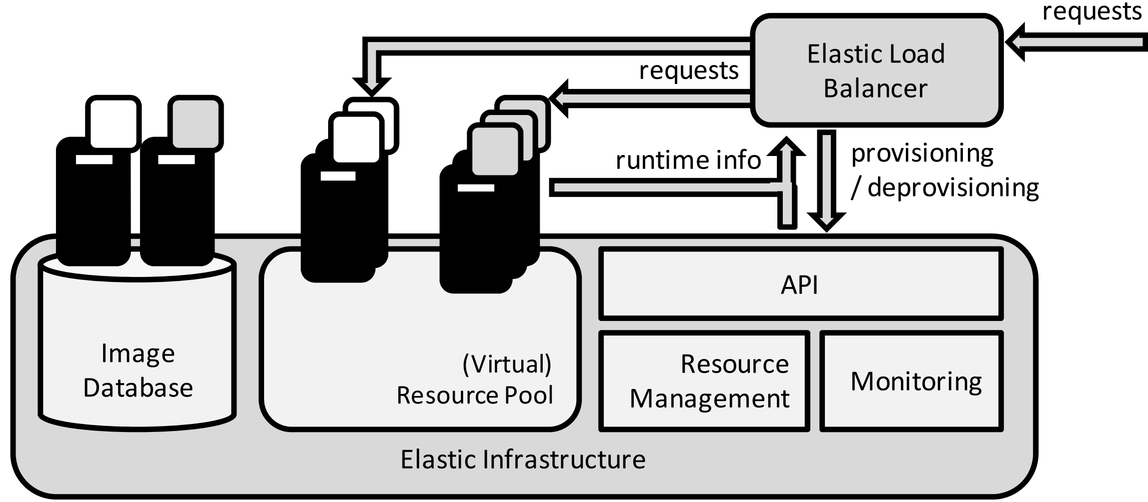
Challenges

To benefit from an elastic infrastructure regarding the enabled alignment of resource numbers to experienced workload, the scaling process of the application has to be automated. Especially, if the elastic infrastructure offers a pay-per-use pricing model, the reactiveness of this process directly affects the running costs of the application. Due to these facts, manual scaling techniques are less suitable. Requests sent to an application are a good indicator of experienced workload and therefore shall be used as a basis for scaling decisions.

Solution

Use an elastic load balancer that determines the amount of required resources based on numbers of requests and provisions the needed resources accordingly using the elastic infrastructure's API.

Sketch



Results

A load balancer determines the number of required resources based on the requests that are passing through it. Each computing node hosting application components is determined to handle a certain amount of requests per time interval. From this property the number of required resources can be computed by the load balancer and is then provisioned on the elastic infrastructure using its API. The number of requests, which can be handled by a computing node hosting an application component, is a crucial design parameter. It significantly affects the effectiveness of the scaling decisions. It should be carefully selected during the design of the application using capacity planning techniques (see below). Also, the elastic load balancer should adjust this parameter during the runtime by

collecting execution times from the computing nodes. Information about the time it takes to provision new compute nodes can also be necessary to deduct effective scaling actions.

Relations to other Patterns

To fully benefit from the monitored compute node properties, the application must use an *elastic infrastructure* (3.1.1).

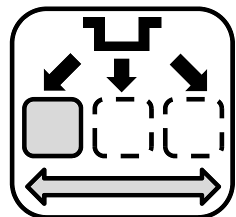
The *elastic load balancer* is most effective if requests are synchronous and thus must be handled immediately. If the application (or some of its components) handle asynchronous requests delaying of some requests to actively influence the workload can be more effective. This is especially the case in less elastic environments such as a private cloud, or if the resource costs of the elastic infrastructure differs over time. Such scenarios can be handled effectively using an *elastic queue*.

Examples / References

By the time of this writing, an elastic load balancer has to be implemented in each individual application and cannot be used as a service.

Capacity planning techniques to determine the number of requests / users a resource can handle are described by [1].

4.2.4 Elastic Queue



How can the number of application components, that are scaled-out, be adjusted automatically based on the number of asynchronous requests in an optimized fashion?

Context

A componentized application uses multiple compute nodes provided by an elastic infrastructure.

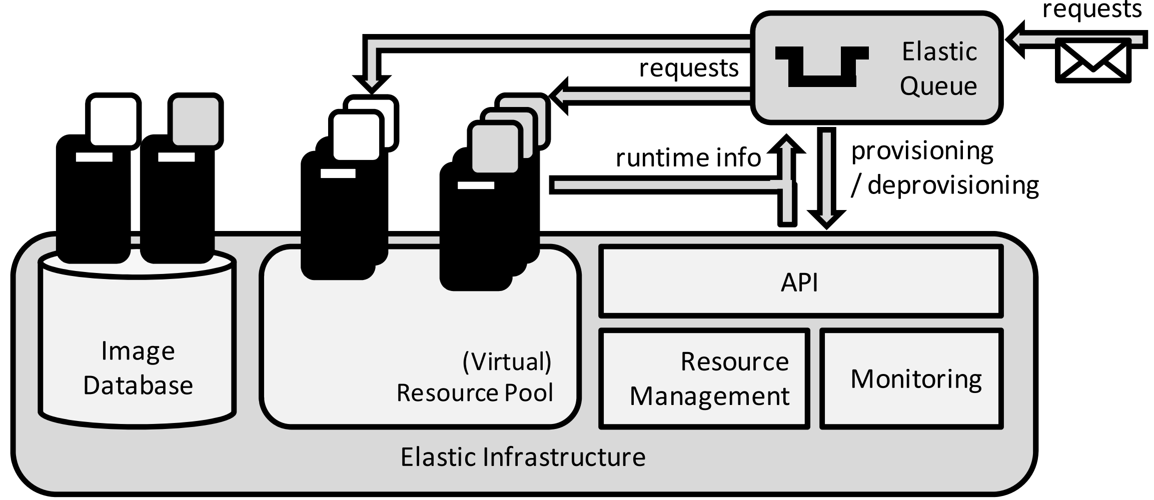
Challenges

To benefit from an elastic infrastructure the number of resources must be adjusted automatically to the current workload. Especially, if pay-per-use is offered the reactiveness of this process directly affects running costs rendering manual scaling approaches less optimal. If the application components handle asynchronous requests additional optimization of the workload execution can be performed by delaying some of them. This can be advantageous, if resources costs fluctuate or the elasticity of the environment is reduced, as in a *private cloud* (2.2.2). In the former case, workload shall be delayed until processing is feasible due to cheap resource costs. In the later case, non-business- critical or time-critical workload, such as report generation, can be moved to times when resources of the private cloud are less utilized.

Solution

Use an *elastic queue* that is used to distribute requests among application components and that scales these components depending on the number and type of messages it contains.

Sketch



Results

Based on the number and type of messages it contains, the elastic queue determines the number of computing nodes to be provisioned. Those computing nodes host the application components that process messages. To speed this process up, individual images for application components are stored in the image database of the elastic infrastructure. The elastic queue can contain different message types that are handled by different components. For example, this could be business critical application functionality and less critical application functionality. Additionally, the elastic queue can respect environmental information, such as the overall utilization of the elastic infrastructure or resource prices. This is used to delay less critical messages by reducing the number of handling compute nodes and to prioritize the business critical functionality if the overall infrastructure utilization is high. Also, the less critical functionality can be delayed, if resources prices are too high to process them at acceptable costs.

Relations to other Patterns

To fully benefit from an elastic queue, the application must use an *elastic infrastructure* (3.1.1).

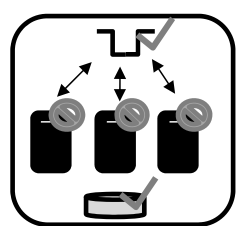
If *low availability compute nodes* (3.1.2) are offered by the elastic infrastructure, the elastic queue can be combined with a *watchdog pattern* (4.3.1).

Examples / References

By the time of this writing, an elastic queue has to be implemented in each individual application and cannot be used as a service.

4.3 Availability Patterns

4.3.1 Watchdog: High availability with unreliable Compute Nodes



How can a high available application be realized using unreliable compute nodes?

Context

An application is build upon a cloud infrastructure that offers a low availability. The availability that shall be assured by the application however needs to be higher, even though it depends on low available resources.

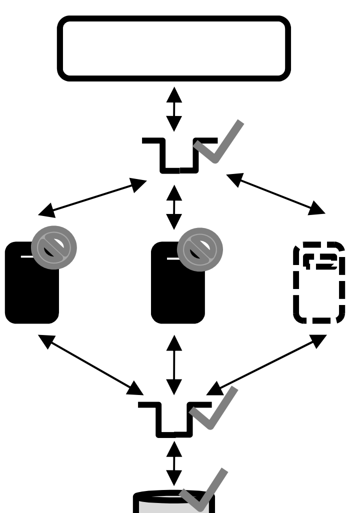
Challenges

Many public cloud offerings only assure a low or no availability of compute nodes. Sometimes, (high) availability is only expressed regarding the possibility to start new compute nodes. To guarantee high availability under such conditions the application architecture needs to be adjusted to enable redundancy and fault-tolerant structures.

Solution

The application architecture is altered to include redundant compute nodes performing the same functionality. High available communication between these nodes is assured, for example by a messaging system. Additionally, compute nodes are monitored and replaced in case of failure.

Sketch



Results

Unreliable compute nodes are used in conjunction with a high available, messaging system or data stores to build high available applications, for example by using a *message oriented middleware* (3.3.1) that offers *reliable messaging* (3.3.2). Compute nodes take a message from their input queue, process it, and put the result in an output queue. These accesses are performed within the scope of one transaction, thus if one compute note fails during the processing of a message, this message reappears in the input queue and can be processed by a different compute node. In conjunction with

an *elastic infrastructure* additional compute nodes can be started in case the load increases or other components fail to assure a constant high availability.

Relations to other Patterns

Low availability compute nodes (3.1.2), *message oriented middleware* (3.3.1), and an *elastic infrastructure* (3.1.1) enable the creation of high available components. In a setting where high available compute nodes are used, the decoupling of components can also increase the performance and enable elasticity.

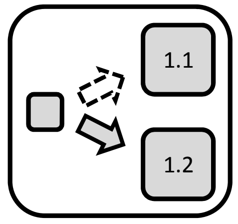
The compute nodes used in this pattern can rely on public, private, or hybrid clouds that may operate regarding the IaaS, PaaS, or SaaS model. Especially, if these clouds offer different service levels the pattern can be used to offer a homogenized quality of service to the outside.

As in every setup where messaging is used, the compute nodes need to consider the delivery assurances made by the messaging systems, such *as at-least-once* or *exactly once delivery*.

Examples / References

Companies like RightScale [70] and ScalR [75] implemented watchdog functionality on top of Amazon EC2 [10]. Their services can be used to manage and monitor virtual servers and include scaling and fault-recovery operations. The watchdog pattern is also described by [27]. How the redundancy used by this pattern affects the overall availability of the application is described by [51].

4.3.2 Update Transition



How can a componentized application be updated, when new versions of application components or the used middleware, operating system etc. become available?

Context

A componentized application that has to be highly available and that uses multiple compute nodes provided by an elastic infrastructure.

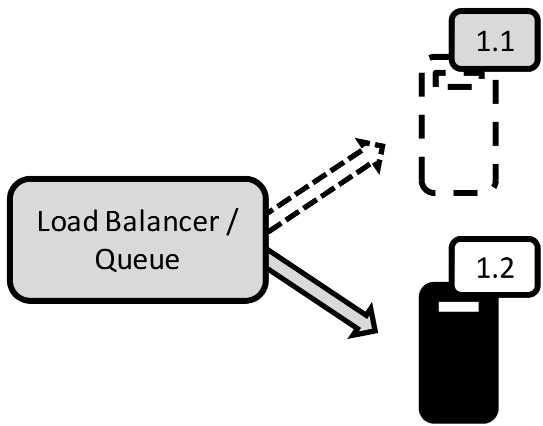
Challenges

If a componentized application offers crucial business functionality, it must not be unavailable during the update to a new application or middleware version. Thus, the transition shall be realized with minimal transition time.

Solution

Provision additional compute nodes that contain the new application or middleware versions additionally to the old versions. Afterwards, shutdown the old compute nodes.

Sketch



Results

Due to the elasticity of the infrastructure and the dynamic provisioning functionality, no running components are updated. Instead, images for compute nodes with the new software version are created and tested. Then, these are provisioned additionally to the old ones. Afterwards those old ones are turned off iteratively. This guarantees a graceful transition between the new and the old application versions.

Additional challenges may arise due to incompatibilities of the software versions. If different versions must not be handling requests at the same time, the transition must be made at once. This is handled by instantiating both application versions independently. The switch can then be made by reconfiguring the access component, such as a load balancer. However, in some cases this can result in a minimal downtime during the transition.

Relations to other Patterns

During the transition between two application versions a magnitude of compute nodes is required that can be handled best by an *elastic infrastructure* (3.1.1).

The complexity of the transition process is also reduced drastically, if application components are *stateless components* (4.1.3) and *loosely coupled* (4.1.2).

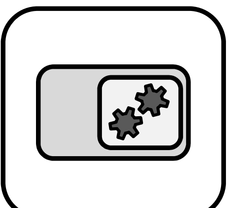
Examples

By the time of this writing an update transition has to be implemented separately for every application to respect the individual order in which application components have to be updated. Windows Azure [63] supports rudimentary transition functionality between a staging and a productive version of compute nodes.

4.4 Multi-Tenancy Patterns

In the following the term tenant is used to refer to a party that uses an application. Multiple users may be associated with a tenant and access the application on his behalf. In this scope, a tenant may be a company that registers to an online application to be used by multiple employees.

4.4.1 Single Instance Component



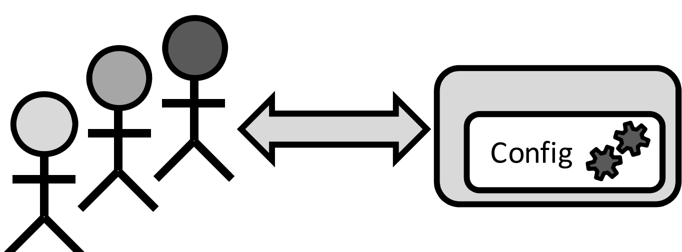
How can an application component be shared between multiple tenants, if individual configuration is not required?

Context
A componentized application is provided to multiple customers.

Challenges
Deployment of componentized applications can be optimized by sharing component instances whenever possible. This is especially feasible for application components that are configured equally for all tenants, such as currency converters. If tenants can share the same resources for such application components, underlying resources can be utilized in more efficient ways.

Solution
Deploy components with equivalent configuration for all tenants only once and share it between tenants.

Sketch



Results
The tenants' individual application instances access the same application component (pool). Therefore, the runtime cost per tenant can be reduced, because the utilization of the underlying infrastructure is increased and the shared component can be scaled for all tenants.

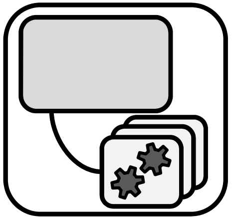
Relations to other Patterns
Even if a component can be shared from the functional perspective, sometimes customer requirements do not allow it. They may, for example, be restricted by laws stating that their application instances may not share any resources with other tenants. In this case, the shared components have to be provided as *multiple instances* (4.4.3).

If some configuration is required by customers but still a common pool of component instances shall be shared, *a single configurable instance* (4.4.2) has to be used.

Examples
Many platform services, for example those used for authentication or user rights management, implement the single instance pattern.

The single instance pattern has been introduced by [64]. In [32] information is given, how the distribution of tenants among such components can be optimized.

4.4.2 Single Configurable Instance Component



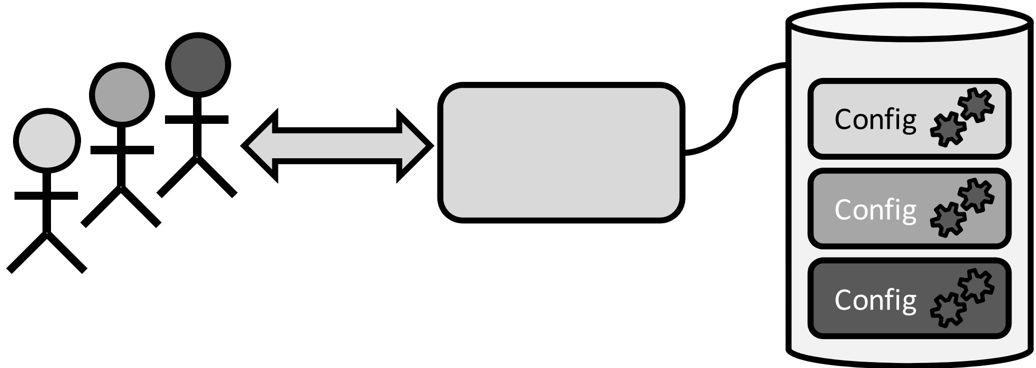
How can an application component be shared between multiple tenants if individual configuration is required?

Context
A componentized application is provided to multiple customers that may configure it.

Challenges
Deployment of componentized applications can be optimized by sharing component instances whenever possible. However, customers often need to apply individual configurations to shared components. An example would be a database server that is shared between multiple tenants, but each tenant may specify an individual database schema.

Solution
Deploy one component that uses individual configurations for each tenant.

Sketch



Results
The component adjusts its functionality according to a configuration that is determined by the tenant accessing it. Therefore, the runtime cost of the component is reduced and utilization of underlying infrastructure is increased, because tenants can share the same component (pool).

Relations to other Patterns
If the configuration is equivalent for all tenants *a single instance* (4.4.1) can be used.

Sometimes, tenants are not allowed to share critical components with other users. In this case a *multiple instance component* (4.4.3) must be used. The same is the case, if the configuration

requirements of tenants differ too greatly. This would lead to an increased complexity of the single configurable component implementation and can also cause performance problems.

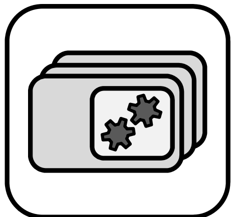
Variations

If the configurations are very small, tenants can send them to the component every time they access it. For example, this is done by web browsers that can inform an accessed web browser about the preferred language.

Examples

The single configurable instance pattern has been introduced by [64]. In [32] information is given, how the distribution of tenants among such components can be optimized.

4.4.3 Multiple Instance Component



How can an application component be provided to multiple tenants who configure it, if sharing is unfeasible?

Context

A componentized application is provided to multiple customers that may configure it and deploy individual instances.

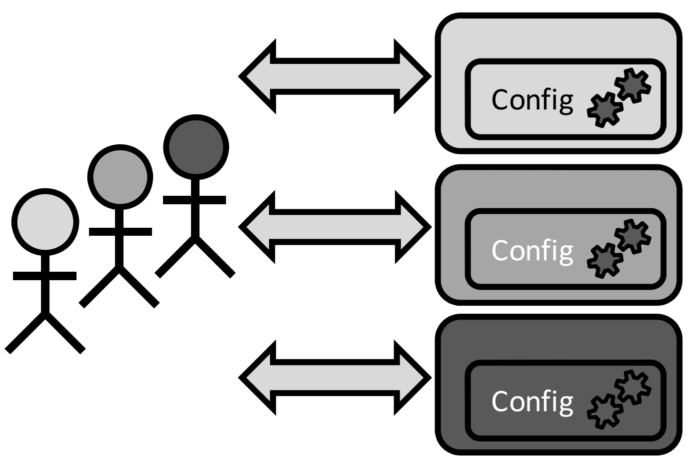
Challenges

If an application shall be instantiated for multiple tenants, some of its components cannot be shared between them. This can be the case due to laws prohibiting tenants to share resources, handling critical functions, with others. Also, tenants may require integration of individually developed application components into the provided application. The application components, management by the application provider, then differ regarding their implementation and configuration.

Solution

Deploy individual component implementations and configurations for each tenant.

Sketch



Results

The deployment of individual component implementations and configurations allows tenants to adjust components very freely. Portions of an application, on which tenants have a versatile behavior, can be realized in such a fashion. However, the application of this pattern hinders resource sharing between tenants.

Relations to other Patterns

Whenever possible a *single instance component* (4.4.1) or a *single configurable component* (4.4.2) should be preferred over the multiple instance component, because they enable component sharing between tenants. This leads to a better resource utilization and reduces the running cost of the application.

Examples

The multiple instance pattern has been introduced by [64]. In [32] information is given, how the distribution of tenants among such components can be optimized.

5 Acknowledgements

The authors would like to thank the following persons for detailed review and inspiring discussions: Tobias Binz, Alexander Nowak, Steve Strauch, and Sebastian Wagner.

6 References

[1] Almeida V.A.F., Menasce D.A.: Capacity Planning for Web Services: Metrics, Models, and Methods, 2002.

[2] Amazon.com, Amazon Web Services. Available at: http://aws.amazon.com/

[3] Amazon.com, Elastic Compute Cloud (EC2). Available at: http://aws.amazon.com/ec2/

[4] Amazon.com: Autoscaling. Available at: http://aws.amazon.com/autoscaling/

[5] Amazon.com: Cloudfront. Available at: http://aws.amazon.com/cloudfront/

[6] Amazon.com: EC2 Running IBM. Available at: http://aws.amazon.com/ibm/

[7] Amazon.com: EC2 Running Microsoft Windows Server & SQL Server. Available at: http://aws.amazon.com/windows/

[8] Amazon.com: EC2 Service Level Agreement. Available at: http://aws.amazon.com/ec2-sla/

[9] Amazon.com: Elastic Block Storage (EBS). Available at: http://aws.amazon.com/ebs/

[10] Amazon.com: Elastic Compute Cloud (EC2). Available at: http://aws.amazon.com/ec2/

[11] Amazon.com: Oracle and AWS. Available at: http://aws.amazon.com/solutions/global-solution-providers/oracle/

[12] Amazon.com: Relational Database Service (RDS). Available at: http://aws.amazon.com/rds/

[13] Amazon.com: Simple Queue Service. Available at: http://aws.amazon.com/sqs/

[14] Amazon.com: Simple Storage Service (S3). Available at: http://aws.amazon.com/s3/

[15] Amazon.com: SimpleDB. Available at: http://aws.amazon.com/simpledb/

[16] Amazon.com: Virtual Private Cloud. Available at: http://aws.amazon.com/vpc/

[17] Apache Foundation, Active MQ. Available at: http://activemq.apache.org/

[18] Apache Foundation: CouchDB. Available at: http://couchdb.apache.org/

[19] Apache Foundation: Servicemix. Available at: http://servicemix.apache.org/

[20] Bernstein P.A., Newcomer E.: Principles of transaction processing, 2009.

[21] Blake M.B., Rosenberg F.: Composition as a Service, 2010.

[22] Brewer E.A.: Towards Robust Distributed Systems. PODOC Keynote, 2009. Available at: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[23] Chang F., Dean J., Ghemawat S., et al.: Bigtable: A Distributed Storage System for Structured Data. 2006. Available at: http://labs.google.com/papers/bigtable.html

[24] Citrix Systems: Xen Hypervisor. Available at: http://www.xen.org/

[25] Cordys.com: Cordys Process Factory. Available at: http://www.cordysprocessfactory.com/

[26] Dean J., Ghemawat S.: MapReduce: Simplified Data Processing on Large Clusters, 2004. Available at: http://labs.google.com/papers/mapreduce.html

[27] Douglass B.P.: Real-Time Design Patterns, 2003.

[28] Enomaly Inc.: SpotCloud. Available at: http://www.spotcloud.com/

[29] Eucalyptus Systems: Eucalyptus. Available at: http://www.eucalyptus.com/

[30] February R.C.: Horizontal Scalable Data Stores. 2010. Available at: http://www.cattell.net/datastores/Datastores.pdf

[31] Fehling C., Konrad R., Leymann F., Mietzner R., Pauly M., Schumm D.: Flexible Process-based Applications in Hybrid Clouds. Proceedings of the 2011 IEEE International Conference on Cloud Computing, 2011.

[32] Fehling C., Leymann F., Mietzner R.: A Framework for Optimized Distribution of Tenants in Cloud Applications. Proceedings of the 3rd IEEE International Conference on Cloud Computing, 2010.

[33] Fielding R.T., Taylor R.N.: Principled Design of the Modern Web Architecture, 2002.

[34] Gilbert, S. and Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 2002. Available at: http://portal.acm.org/citation.cfm?id=564601

[35] Google Inc.: Google AppEngine. Available at: http://code.google.com/appengine/

[36] Google Inc.: Google Apps for Business. Available at: http://www.google.com/apps/intl/en/business/index.html

[37] GSA, Apps.Gov. Available at: http://apps.gov

[38] Hohpe G., Woolf B.: Enterprise Integration Patterns, 2004.

[39] IBM: CloudBurst. Available at: http://www.ibm.com/ibm/cloud/cloudburst/

[40] IBM: DB2 Software. Available at: http://www.ibm.com/software/data/db2/

[41] IBM: DeveloperWorks RESTful Web services: The basics. Available at: https://www.ibm.com/developerworks/webservices/library/ws-restful/

[42] IBM: LotusLive. Available at: http://www.lotuslive.com/

[43] IBM: Systems z. Available at: http://www.ibm.com/systems/z/

[44] IBM: Tivoli Provisioning Manager. Available at: http://www.ibm.com/software/tivoli/products/prov-mgr/

[45] IBM: Tivoli Service Automation Manager. Available at: http://www.ibm.com/software/tivoli/products/service-auto-mgr/

[46] IBM: WebSphere Enterprise Service Bus. Available at: http://www.ibm.com/software/integration/wsesb/

[47] IBM: WebSphere MQ. Available at: http://www.ibm.com/software/integration/wmq/

[48] IBM: WebSphere Software. Available at: http://www.ibm.com/software/websphere/

[49] Intalio Inc.: Business Process Management. Available at: http://www.intalio.com/bpm

[50] Krafzig D., Blanke K., Slama D.: Enterprise SOA, 2007.

[51] Leymann F., Roller D.: European Patent EP0965926 Improved Availability in Clustered Application Servers, 2005. Available at: http://www.europatentbox.com/patent/EP0965926A3/abstract/1178894.html

[52] Leymann F., Roller D.: Production Workflow: Concepts and Techniques, 2000.

[53] Leymann F.: Cloud Computing: The Next Revolution in IT, 2009. Available at: http://www.ifp.uni-stuttgart.de/publications/phowo09/010Leymann.pdf

[54] Mell P., Grance T.: The NIST definition of cloud computing. National Institute of Standards and Technology, 2009.

[55] Microsoft: Hyper-V Server. Available at: http://www.microsoft.com/hyper-v-server/

[56] Microsoft: Office Live. Available at: http://www.officelive.com/

[57] Microsoft: Overview of the Windows Azure VM Role. Available at: http://msdn.microsoft.com/en-us/library/gg433107.aspx

[58] Microsoft: Sharepoint Designer. Available at: http://sharepoint.microsoft.com/en-us/product/related-technologies/pages/sharepoint-designer.aspx

[59] Microsoft: SQL Azure. Available at: http://www.microsoft.com/sqlazure/

[60] Microsoft: SQL Server. Available at: http://www.microsoft.com/sqlserver/

[61] Microsoft: Windows Azure AppFabric Overview. Available at: http://www.microsoft.com/en-us/appfabric/azure/middleware-services.aspx

[62] Microsoft: Windows Azure Storage. Available at: http://www.microsoft.com/windowsazure/storage/

[63] Microsoft: Windows Azure. Available at: http://www.microsoft.com/windowsazure/

[64] Mietzner R., Papazoglou M.P., Leymann F.: Defining composite configurable SaaS application packages using SCA, variability descriptors and multi-tenancy patterns, 2008. Available at: http://portal.acm.org/citation.cfm?id=1381304.1381988

[65] NewServers: Bare Metal Cloud. Available at: http://newservers.com/

[66] OASIS: Web Services Business Process Execution Language Version 2.0, 2007. Available at: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[67] openQRM Enterprise GmbH: openQRM. Available at: http://www.openqrm.com/

[68] Oracle: MySQL. Available at: http://www.mysql.com/

[69] Oracle: Oracle Database Software Downloads. Available at: http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html

[70] RightScale Inc.: RightScale Cloud Management Platform. Available at: http://www.rightscale.com/

[71] Rosenberg F., Leitner P., Michlmayr A., Celikovic P., Dustdar S.: Towards Composition as a Service - A Quality of Service Driven Approach, 2009. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4812599

[72] RunMyProcess: Workflow Builder. Available at: http://www.runmyprocess.com/

[73] Salesforce.com Inc.: CRM & Cloud Computing. Available at: http://www.salesforce.com/

[74] Salesforce.com Inc.: Force.com. Available at: http://www.salesforce.com/platform/

[75] Scalr: Scalr. Available at: https://scalr.net/

[76] Varia J.: Architecting for the Cloud: Best Practices. Whitepaper, 2010. Available at: http://jineshvaria.s3.amazonaws.com/public/cloudbestpractices-jvaria.pdf

[77] Varia J.: Cloud Architectures. Whitepaper, 2008. Available at: http://aws.amazon.com/articles/1632

[78] VMware: vSphere Hypervisor (ESXi). Available at: http://www.vmware.com/products/vsphere-hypervisor/

[79] Vogels W.: Eventually Consistent. 2008 Available at: http://portal.acm.org/citation.cfm?doid=1466443.1466448:

[80] W3C: Web Services Description Language (WSDL) 2.0, 2007. Available at: http://www.w3.org/TR/wsdl20-primer/, http://www.w3.org/TR/wsdl20/, http://www.w3.org/TR/wsdl20-adjuncts/

[81] Weerawarana S., Curbera F., Leymann F., Storey T., Ferguson D.F.: Web Services Platform Architecture, 2005.