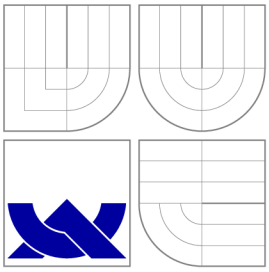
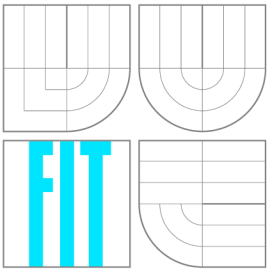


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# CLOUDOVÝ APLIKAČNÍ RÁMEC TYPU INFRASTRUKTURA JAKO SLUŽBA

CLOUD FRAMEWORK ON INFRASTRUCTURE AS A SERVICE

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. DAVID PECH

AUTHOR

VEDOUCÍ PRÁCE — Ing. RADEK BURGET, Ph.D.

SUPERVISOR

BRNO 2013

# Sem vložte zadání

# Abstrakt

Práce se zabývá podrobnou analýzou požadavků na moderní aplikační rámec pro prostředí cloud. Za pomoci standardních návrhových vzorů a technik připravuje teoretický základ a pravidla, která musí uvnitř rámce platit. V práci je realizována referenční implementace a připravena demonstrační aplikace středního rozsahu, aby představila výhody plynoucí z užití frameworku.



# Abstract

The thesis covers an in-depth analysis of the requirements for a modern application framework that runs in the cloud environment. It uses standard design patterns and approaches to prepare guidelines for the framework. A reference implementation is created to prove framework concept. The medium-sized demo application is also developed to prove the framework benefits.



# Klíčová slova

cloud, aplikační rámec, design aplikačních rámců, distribuovaný systém, vrstevnatá architektura, infrastrutura jako služba

# Keywords

cloud, framework, framework design, distributed system, layered architecture, infrastructure as a service

# Citace

David Pech: CLOUD FRAMEWORK ON INFRASTRUCTURE AS A SERVICE, diplomová práce, Brno, FIT VUT v Brně, 2013

# CLOUD FRAMEWORK ON INFRASTRUCTURE AS A SERVICE

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Radka Burgeta, Ph.D.

.....

David Pech

10. května 2013

## Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Radku Burgetovi, Ph.D. za jeho odbornou pomoc při zpracování tohoto tématu.

# Contents

# Chapter 1

# Introduction

A framework design has always been a complex activity that can dramatically influence a product time-to-market aspect. A sophisticated framework with solid theoretical background can increase an application quality and significantly decrease its development time.

This thesis has a goal to produce a theoretical background to create cloud frameworks that support rapid application development. The selected technologies are not the only possible implementation options, on the contrary the framework design suggests to use others components in integration.

The result of the practical part is a reference implementation in a selected technologies and a simple yet powerful application that demonstrates the benefits available when using the framework ideas in the real-world project.

The framework will strongly suggest to use a domain driven design (DDD) [1] as a core part and provide extensive support to minimize the effort related to adopting these concepts. The DDD has numerous advantages over traditional entity-level paradigm.

This thesis won't create a massive over-engineered framework that tries to solve every domain problem for a developer. The result will more likely be a minimalistic framework that is easy to integrate to any technology stack and that enables a real-world project to move from standard server hosting solutions to the cloud with all the offered benefits.

The first part of thesis focuses on analysis of the cloud environment and requirements collection. Chapter two advocates the need for a new framework solution and depicts the current situation in the cloud framework market. There is an elaborated analysis on a cloud user types in the chapter three. Chapter four discuses in-depth the characteristics of the cloud from a technical and technological perspective. In chapter five framework goals and requirements are defined based on the previous observations.

When all the requirements are collected, the framework can be built in the second part of this thesis. In chapter six there is a thoroughly described process of designing the framework that matches all the requirements. A technology is evaluated and selected for a reference framework implementation in chapter seven. Chapter eight contains a detailed

look at the implementation process.

Chapter nine describes a development of a demo application based on the framework along with all the pros and cons that have been met along the way.

In the last chapter an evaluation of the framework capabilities is being made and the thesis concludes in retrospection.

# Chapter 2

# Motivation

There are too many different frameworks currently at use at the software industry. Creating another one needs a justification.

Majority of frameworks are built up from a so-called personal library that experienced programmers used to have in the past. [2] Current frameworks are created by companies as a side effort apart from their main product. These frameworks start as a simple set of very specific classes that are generalized in time. The framework has to solve a simple problem domain at first. Later as the framework is getting wider audience and the community is growing, the features of the framework also grow.

On the contrary, some frameworks are designed from scratch with few transparent guidelines and ambitions. Only a small amount of them is engineered without connection to a large start-up project, however. The second group is often easier to use in other projects than the first, because guidelines keep frameworks more coherent and generally usable. [3]

Many frameworks are older than ten years and are still deployed among starting projects. The modernization process is the key element of success in current highly competitive software industry.

Even the best frameworks tend to age. Of course they can still fulfill most of daily task currently requested, but their design and guidelines are getting rusty. The community is expecting certain behavior from the framework and to significantly change this behavior is not possible in the most cases.

I believe that the cloud concept is so distant from traditional single-machine or peer-to-peer application development that it needs to completely rethink the application structure at fundamental level. For this environment a new framework is needed.

Of course the solution may be to find an existing product. This most certainly doesn't apply to all the situations, in fact the vast majority of simple software tasks have been already solved, it's just about finding the correct of many different solutions.

## 2.1 Available Cloud Frameworks

The main characteristics of such frameworks are their ability to extend computation seamlessly across volatile number of computation units. These implicate complex questions that have to be dealt with:

- No traditional file-system - often programs can't directly work with files and an abstraction layer is used
- Module or other computation unit boundary - program is divided into parts which can be executed in parallel
- Failover, HA - framework supports means to achieve high fault-tolerance
- Ease of deployment - it must be easy to replicate and install a new clone of the framework environment [4]

### 2.1.1 Proprietary Cloud Engines

The most notable representative is the Google App Engine [5]. The engine has a large amount of supporting staff both from the company and also from open-source world. The framework is aimed to support rich internet applications.

The largest disadvantage is the proprietary runtime environment. The GAE runs on Google web servers, uses Google services and application cannot be fined tuned to use extensively unsupported technology.

Similar situation is with Microsoft that offers an application hosting services on their technology stack [6].

### 2.1.2 Extending Traditional Frameworks

Every single machine framework can be extended to support multi-machine environment up to some level. But these frameworks have other goals and the cluster support means bending their internals up to some level.

The support may be developed, but if it is not a core part of a framework, it may be hard to use or prone to synchronization issues.

### 2.1.3 Cluster Computational Frameworks

A Hadoop [7] can be chosen as a typical representative of this category. These frameworks are suited for massively parallel computation across a large server cluster. Each node executes the same program and operates on the similar type of data as all the other.

The primary goal is data transformation to some other format or data type [8].

These solutions can be extended and used in server application, however their installation and maintenance requires an expert on these technologies. They are not suitable for rapid application development where the learning curve and entry time of a new project developer must stay low.

## 2.2 Yet Another Framework

Of course there are too many frameworks already for any individual to summarize them or to understand them all. I believe that it's a necessity of introducing a new one because its aims are dramatically distant from the current standard framework representatives.

The framework creation will be started in this thesis, but it will be designed with the best practices in mind, so the development may continue later on.

## 2.3 Complex Design Problem

The quest for a quality cloud framework is a complex engineering problem. The focus can't be drawn only to the technical solution itself. The computation force to handle multi-computer environment already exists and this thesis is not here to question the fact that even in an assembly language the application can theoretically exists.

The main goal of the thesis is the concept, theoretical foothold for the framework. It's aimed as a manual for building certain types of cloud applications.

The programming code is not as important as the documentation, examples of its usage and other support material that developers will use to learn.

## 2.4 Reinventing the Wheel

Numerous theses on a framework design exist. [9] [10]

Their main goal is to evaluate the current area of knowledge, pick best technologies according to some criteria and design a framework that can hold the technologies coherent and in one place.

I strongly believe that the technology selection is the choice of a developer. As the application is offered to customers, changes need to be made constantly in order to keep up with changing requirements. The change of essential storage or any other technology is inevitable for the most software products that live through few years.

One of the framework goals has to be an ability to support existing enterprise-level frameworks for integration. This framework will do only one thing (cloud support) and it won't interfere with developer specific technology choices.

# Chapter 3

# Public Cloud Environment

The definition of the public cloud environment is rather vague due to the number of use cases and purposes the cloud can serve. I find the best way to describe the cloud environment to sum up all the important areas[11]:

- *Internet Connection* - the cloud is always accessible from the Internet and in the most cases it's the main entry point of the application
- *Real-time Scalability* - number of required resources may change during each month or even during one day as the traffic changes
- *Fault-tolerant Environment* - no matter faults in hardware the application should be running
- *Monitoring* - the complex system requires monitoring because the failures will always occur and it's vital to locate them as soon as possible [12]
- *Automated Deployment* - under these conditions it's not possible to deploy an application by hand. All the procedures must be automated to some level to prevent errors and misusage [4]
- *Mirrors (Clones)* - developers have one or more environments set up exactly the same way as a production one for testing purposes
- *Integration with Other Systems* is in many cases the essential and primary service the application is providing
- *Automated Testing* is essential to keep up the quality of the product [13]

## 3.1 Domain Boundaries

To narrow the boundaries of this thesis only the public cloud environment with its specifics will be described.

The environment is opened to public and provides some kind of service. The security levels must be designed accordingly.

It's important to mention that many cloud services are based on simple cloning (profile, clones...) of one application. The Software as a Service [14] will be explained in detail.

### 3.1.1 Cloud Advantages

Cloud environment offer large number of benefits, to name some of them [15]:

- It's *modern*. Everyone wants to advertise that his IT is moved to the cloud, customers react to this fact positively.
- No need to solve *problems with the physical hardware*.
- *Lower Running Costs* - studies show [16] that overall cost for this type of hosting is lower than other hosting solutions.
- *Service Ecosystem* - offered services are tuned to work seamlessly together.
- *„Unlimited" Resources* - many of the offered services don't pose any limit on the maximum storage capacity etc. In reality there must be a level, but it is so high that it is not a real problem.
- *Deployment and Monitoring Support*
- *Support Services* - each large vendor offers a large number of secondary services adding a large business value to the basic virtualization technology. Simple databases, block storage, management tools or monitoring tools are good examples of such services.

### 3.1.2 Cloud Disadvantages

Cloud vendor most commonly provides a technical solution of a virtual machine. Of course this seems appropriate and may work well in a lot of cases, but there are certain scenarios that this may seem as a disadvantage.

- *Constant application load is not possible* - it's difficult to predict behavior of your „neighbors" in the cloud. Other companies who operate on the same physical machine can do a lot of intensive operations and this has drawbacks for your application. The number of the CPU cores (or other „computational" units) is constant and has the same performance every time. But the hard disk drives, network, entropy and other physical resources are not necessarily dedicated to your application only. This is the place where your application performance can drop, because every real-world application has to rely on network or HDDs to work.

- *Artificial Limitations* - certain providers impose artificial limits that should improve certain nontechnical areas. For example Amazon Web Services (AWS) limits the number of public IP addresses and even bills every hour of unbound IP address. This should suggest reasonable housekeeping, but on the other hand it's completely artificial and not a technological limitation.

- *Pay As You Go* - for most situations it's a benefit to pay only for the resources that you have used, but the largest amount of the server environment is often an initial investment to buy and configure the server. If you already own a server, it's already running, other solutions may seem a lot cheaper (dedicated server housing etc.). Of course a risk analysis must be made (failing hardware) to prove this point.

## 3.2 Manager's and Business Perspective

As for the manager the term cloud is most certainly an interesting aspect of the software system. Given the current buzz of the word, everyone wants to have his data in the cloud and a lot of customers is asking if the solution is based on the cloud. This can be used as an advantage in such a highly competitive field as IT is.

The manager must watch closely the complexity growth:

- Are the developers on the project skilled enough to build the application?

- Will we maintain the quality of our product and keep our deadlines?

The only correct solution to this is of course hiring a skilled coach or consultant who already has a lot of experience with the cloud. The questions are more difficult to answer than in the traditional applications.

There is less space for mistakes. The cloud opens one day, the website becomes public and the application is put into the pilot mode. If a hurricane hits the datacenter that day, many of the customers are disappointed that the application isn't working and may never come back. When you develop a traditional application, package the version 1.0, you can test it for days before you release it in public.

An unexpected success may happen and suddenly much more customers want to use your application. The infrastructure may not be ready yet. The same problem applies for the traditional web applications also. The cloud should have much simpler solution - renting more CPU time, virtual machines and so on. Also many cloud providers offer flexible storage capabilities so migration to a larger disk array may not be a problem.

A hardware hosting and the vendor security trademark are valuable assets among customers. Especially the US citizens are greatly concerned about the security of their data, so it helps the brand to use cloud data center of the large well-established company.

Working with a community is a significant benefit for cloud and web oriented companies. The community can often provide support for new users, so the first level support can be reduced. When the community mass reaches some point it becomes self-sustained and the company can significantly reduce investments in this area [17].

The user community looks out for changes. When the module is released, everyone learns its functions and how to control it. When the change comes later on it may enhance the environment, but it's a change and people often resist changes in general.

## 3.3 Developer's Perspective

The great benefit for the developer is a short release cycle. With each stage of a development, a version is published and the customers are accustomed to changes in the cloud environment. It's natural to release a version every month, sometimes even every week. The changes are not drastic as a new major version comes up, but the application is incrementally getting better. Because of the risks involved the number of any major changes to the cloud is limited [13].

Things can always work better and this is completely true the case in the cloud. Given its complexity there are always areas to improve as the application is working 24/7.

The technologies might have some restrictions from their standard installation setup. The security is generally higher in a cloud environment and numerous extra security policies are applied. Some thresholds may prove to be too low and there might be no way to increase them. It is always an advantage to develop in suggested patterns and don't bend any of the technologies to your own needs.

Special attention must be paid to the deployment process. It must be automated to the highest possible level. It is the cloud the availability that is always an issue and every outage should be minimalized. A number of such incidents should be minimized, too. A good approach to reduce these times is to better understand the wiring of the cloud components. There may be dozens of interconnected components that seamlessly work together and provide final service for the customer. The interconnection is the place that can, up to certain point, help to hold fluctuations until the component is restarted or updated. All the communication is held until it is online again. Only if it is inactive for a longer period of time, an error is thrown.

The developer's point of view is not as straightforward as the previous one. The application is built from scratch very rarely and this implicates a very different attitude toward the cloud environment.

### 3.3.1 Migrating Legacy Application

The cloud applications may be built the traditional way to run on a single node using standard single-node database. In this manner the cloud benefits are greatly reduced. For an established web application even a migration to such a schema could be challenging enough, but it's a large promise for the future as different parts of the application are migrated.

It's crucial to progress iteratively. When the application is migrated „as-is", one phase is completed. A next step may be migrating the application files from file-system to specialized, replicated and highly available storage the cloud provides. Each step must be done with caution and never in parallel.

This process is of course much slower than writing a next generation of the application that supports the cloud technologies in principle, but is used much more often due to its conservative nature. Any migration step may prove wrong, it must be easy to downgrade the version, migrate back to previous well-working solution.

Also the developers tend to learn new technologies bit by bit and must not be overwhelmed.

### 3.3.2 Writing Application from Scratch

It's a challenging opportunity for any developer to build a scalable application. It's much more challenging to build scalable and high-available application. Every area of cloud environment can be subject of study itself. The developers must design the application with priorities in mind. It's not possible to design every aspect to the best of the developer abilities and still not miss the deadline. The knowledge is evolving by the process and the product has certain budget to fulfill.

## 3.4 IT Maintainer's Perspective

A monitoring is the main tool for an IT technician who keeps the cloud application running. Collecting statistics also help especially in the long run assessment.

The monitoring should be built along the automated software that is able to keep track of thousands of little checks every hour and reports any inconvenience to the maintainer. Vast majority of check is of the technical nature testing cloud condition in ways like CPU load, bandwidth usage etc [12].

Statistics collection is crucial for long-term strategy assessment. Without them no proper action can be chosen. Statistics from the usage of the application itself are valued among the business people who can cut their offers exactly for certain customer.

# Chapter 4

# Cloud Infrastructure Composition

From the technical point of view, clouds can be classified in multiple ways.

## 4.1 Cloud Classification

### 4.1.1 From the Ownership Point of View

This classification has implication mainly on the target audience who uses the cloud [11].

- *Private* - large companies have so demanding requirements that the cloud solution is best fitting for them. These clouds often have less important UI and user experience is not one of the priorities as it is used mainly by trained employees.
- *Public* - a cloud offering public services. Cloud can provide cloning or copying of the single application instance.
- *Hybrid* - combination of both previous. A lot of hybrid cloud can be seen as evolved company information system that has been enhanced to serve company customers as well.

This classification is obvious and suggestions about the used technologies and priorities of the different clouds are its real impact. There is no solid technological or any other boundary between the solutions and if a private cloud becomes partly visible to the Internet, it suddenly is a hybrid model.

### 4.1.2 From the Technical Point of View

This classification takes into account the aspect of technical solution of the application. Being in the cloud can roughly equal to having several virtual machines that your application is running on. The classification divides the cloud by the level of access to the virtual machine (or computational unit) that the cloud provider offers.

These types of cloud differ also by the type of customers.

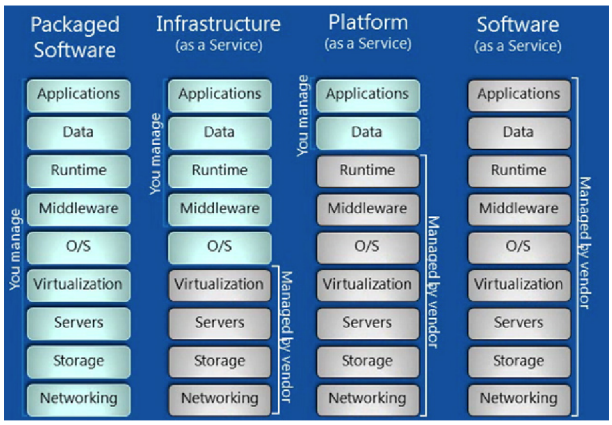



**Figure 4.1:** IaaS, PaaS, SaaS Comparison [18]

### 4.1.3 Infrasture as a Service (IaaS)

IaaS can be seen as the lowest level of cloud virtualization possible. In a typical scenario an administrator root access is granted to the whole virtual machine. So it's possible to install any type of application that can be run on the architecture given by the CPU type.

This approach implies a high level of administration, it's necessary to install entire application infrastructure which can span across many virtual machines. The main limitation may be the OS kernels which are prepared by the cloud providers.

Almost every cloud provider offers custom application extensions to most common tasks solved on the cloud. These often include lower pricing than for running the service on your own in computation time. The common services may have broad range:

- *Load balancer, IP address switching* - this feature can save a lot of money because load-balancer is micro component which has to be always running and consumes low amount of resources.
- *SQL DB Equivalent* - custom SQL DB engine providing most common functionality up to certain SQL standard. Simple use-cases may contain relation DB schema creation, populating DB with data etc. As the application is getting larger it may be limiting because these custom engines have limited level of tuning and it's difficult to debug them given the „neighbor" traffic in the cloud.
- *NoSQL DB Equivalent* - suitable for no-sql DB using different documents as the central objects and map-reduce as the computation method. Standard NoSQL DBs have fewer functions but can execute queries faster than traditional SQL DBs. Some cloud providers suggest using them over SQL DBs.
- *File Storage* - using file-system on the virtual machine is always more expensive than using a custom file-storage service. These storages are often paid by the real amount of data used and have better support with backups.

- *Backups, Archiving Tools* - storage service for write-only data that are very rarely read. If the service is provided, it's always cheaper that file storage itself.

Payment models are based on the services used in the cloud. The general rule is that a virtual machine with certain amount of memory and CPU is rented and charged for its running time. This is the most expensive item, other services provided should be significantly cheaper as the providers want to motivate their customer to use them for two main reasons:

- *Cheaper* than general-purpose CPU time and self-installed solutions
- *Binds* the application to the cloud provider

Currently the service is provided by: Google Compute [19], Amazon EC2 [20], Windows Azure [6] and many others.

### 4.1.4 Platform as a Service (PaaS)

In this scenario the execution environment (platform) is given. Typically this includes stack created by large software house from its various products. It's harder to generalize because platforms significantly differ [21].

Execution environment is often considered a virtual machine running only certain byte-code. It can be an application framework that developers are bound to use. The database technology is given by cloud provider preferences and there is often no other choice.

Payment models are more specialized counting various resources used by the cloud user. The possibilities are more limited than in IaaS so the pricelist can be more precise. Cloud provider offers can be better targeted because it's a lot easier to offer a DB specialist to enhance your DB than an IT technician because your virtual machines have a high load.

Currently the service is provided by: Google App Engine [5], Amazon [22], Windows Azure and many others.

### 4.1.5 Software as a Service (SaaS)

SaaS is on the opposite side of the spectrum and represents for the user a specific application. In many cases there are minimal differences between cloud application and public-hosting web-application from the end-user viewpoint.

Software in this case is a private copy, clone or instance of certain application. The instance can be customized to certain limits that are very hard to break. Users who need changes in the standard cloud instances are often offered a made-to-measure development

of custom instance. This scenario implicates developing custom information service (IS) serving customer needs that is originated in the cloud instance.

Payment models are based on the number of users, licenses or resources that end-user has. To attract the most end-users as possible, it has become quite standard to offer a limited trial version of the instance. It may be limited for amount of time or resources used, but the end-user must have an opportunity to see the application in action and evaluate it.

A chain of providing is often applied as a company develops an application that is deployed on IaaS virtual machines and is offered also to end customers in the form of SaaS.

### 4.1.6 Choosing the Correct Cloud

The decision, which type of the cloud to choose, must be long-term and very cautious. The environment most similar to the server hosting is IaaS. This may be preferable if a company migrates already existing server installation etc.

Using PaaS or specific provider services on IaaS in the application always creates some level of dependency that has to be considered carefully before doing so. It may be very hard to change the service later on as standardization process in this area is not optimal.

## 4.2 Storage vs. Computational Separation

The main technological innovation of the cloud is the concept of clean separation between storage and computational facility. It can be regarded as a parallel to programming level evolution when a program code was separated from the data it has computed with.

We could already use two hosts, one with a database and the other running an application. At this level a lot of problems arises:

1. The application / database performance may not be sufficient - this problem can be addressed by scaling either horizontally or vertically. Both approaches require a significant investment like buying more hardware or hiring more labor to develop the application.

2. Hardware failure - the server has to be restored to certain backup point and the data must be restored up to the most recent backup. Of course a more robust solution like replication, master / slave concept or failover may exist but this dramatically complicates the infrastructure.

The innovation is in the level of separation the cloud offers. On the storage part they provide is high-available, failover-enabled storage facility that is online at any point

and that has no real possibility of losing data. Strictly speaking these parameters can be expressed in availability so high that it is in the margin of few seconds timeout per year and probability of disk storage failure so low that could happen once in a millenium. Without any extra setup these resources are ready at your disposal.

The computation part is measured as computation units. These are highly technology dependent, best synonym may be a CPU core. Cloud providers often provide some level of comparison chart that one unit is roughly equivalent to certain CPU. A virtual machine is composed of virtual disk that is small enough to contain the virtualized system and your application only. The virtual disk image is also loaded from a storage, so the computation part is composed only from certain amount of computation unit and memory that is available to the virtual machine. The memory may be even partly shared among several virtual machines with modern virtualization technologies so the requirements for the real physical server are even lower than simple sum of the virtualized machines.

As a cloud user you can select a virtual disk image and run it with any number of computation units and memory that provider offers. If the memory is not large enough, no problem really exists, you can just shut down the machine and start a new virtual server with more memory. Customers can easily scale their needs with this model. With modern technologies it's not even required that the physical processor, memory and RAID storage are anywhere nearby.

The innovation lies mainly in the simplicity and short response time that cloud providers can offer for scaling machines. This approach would be several levels more difficult without their technologies.

## 4.3 Cloud Friendly Technologies

There are quite a few requirements for a technology to become usable in the cloud environment:

- *Mature technology with community* - no experimental projects are allowed in the cloud.
- *High-availability support* - to a certain point, the service must be online for the most amount of computation time and outages longer than several seconds are not tolerable.
- *Replication, failover, redundancy, online backups* - all these criteria aim towards conservation of the data in the event of failure.
- *Fine-grained security management* - the environment must be completely separated as in hosting services.

- *High performance requirements* - any operation must be completed quickly to serve customers in the cloud. Techniques to move computation toward DB or storage system are very limited. There are no equivalents to stored procedures or server functions.
- *Highly concurrent access* - resources can't block each other and cause deadlock or other parallel disastrous situations.

A lot of traditional technologies is problematic when they are used in the cloud. The context in which they have evolved was quite different and requirements were mainly to serve on one machine and provide resources for one or two applications at most.

Thus new concepts have been created by the long-term evolution. One of the milestones in the evolution understanding is a success of large social networks and other modern media. They often provide fairly simple services from the technical point of view:

- Storing status messages or short „tweets“
- Uploading photos
- Maintaining your own profile
- Connections to other profile in the social network
- Publishing various types of events, map planning support etc.

The task to store basic data is simple enough, in RDMS could be realized by several interconnected tables. But the main problem with this solution is scalability. Every profile is self-sustained with most of the data (except the links to other profiles). Social media can be used by hundred millions users and this amount of data is too high to handle for traditional single-computer storage systems. Given other requirements like high-availability, the whole concept of saving data at this scale must have been rethought from scratch.

### 4.3.1 SQL Databases

These databases don't meet cloud application requirements in general. The ACID transaction model is the major cause of the problem, but is certainly not the only one hard to overcome.

**Cloud Limiting Concepts** The ACID approach keeps the SQL system performance low. Separation of transaction environment needs a high level of supervision and it is hard to parallelize. There are several types of transactions that differ by the way the level of separation offered. The simple approach is to use serial processed transactions. This is a

complete parallelization killer. Other transaction level are trade-offs between parallelism and data consistency. In small system this may not be an issue, but in a large database environment this often leads to aggregation query inconsistencies. If the application is failing at this database layer, it may be disastrous and as any other synchronization problem extremely hard to fix reliably [23].

The logical conclusion is to move every write intensive operation to the database layer to minimize data fluctuations. This breaks application because stored procedures are low-level and business logic is high-level. This approach fragments the application and may cause premature optimization [24].

On the other hand if stored objects don't include inner collections, the SQL read performance and query optimization engine can be used with a benefit. Database objects having inner collections breaks the whole join-table concept and for most cases a complex ORM is required to reconstruct collections properly.

Consitency problems are another aspect of using SQL databases. The storage of interconnected-table object has to be handled in transaction. But sometimes it is inevitable to make by-hand custom queries to the database. This could happen as a correction to failed migration, updating records in the application can't handle etc. Let's imagine classical approach with table A that needs a join to table B. The relationship is 1:1 and given other constraints two separate tables must exist. So you put a foreign key to table B that refers to the table A. If you delete a records in table B, the link from A is missing and there is no way to enforce this requirement in traditional SQL database. The stored database objects are complex in the real-world applications.

**Possible Usages** The benefit of SQL approach to database storage is a rich query system. The database indices can be heavily used and data retrieval can be extremely fast for certain data types. The SQL system is well tested and has been taught for decades, so many developers are familiar with it. When your company switches to the cloud, there are many new technologies to be learnt only to migrate the application without touching the source codes. So using well-known technologies could be a great benefit to the team.

The cloud SQL databases are in general simpler than their single-instance alternatives. A lot of tuning query types is not available and have been removed as a price for a cloud hosting unification. This should not pose a problem if the SQL database is not bent against its original purpose - storing user-related data only.

By its computation model SQL databases store virtually any type of data. But they are not suitable to store statistics, heavy-write data models or structured objects. There are major cases that should motivate developers to look for data storage elsewhere.

### 4.3.2 No-SQL Databases

No-SQL has become an umbrella term for most modern databases which do not use the concept of the SQL for data storage. The data storage engines vary enormously as their purposes differ significantly.

**Document Oriented Storage Systems** A large group of No-SQL databases is well suited for saving structured object-oriented objects. The storage unit is called a document, internally is composed from fields, arrays, hash-objects and values. This is well-suited for most object-oriented applications that need to store some type of data transfer objects (DTOs).

On the other hand one major disadvantage exists - collections in which the documents are stored can't be linked with collections on the database level. The document oriented storage implicates this principle, but it's hard to overcome for a developer migration data from SQL database where everything can be joined in one query. The database link exists, but is maintained by the application itself.

The map-reduce computation model is used for advanced queries. In theory this model has same computational strength is the one of the SQL, so no major problem arises.

The document storage unit is the key to better performance of the DOS. There are no transactions and updates are always single-document level only. This leads to the eventual-consistency model which is different from the SQL.

The document storage logic would not be useful as it would represent the records of the same size as the records in the SQL approach. The document is much richer and contains much more data and the document must be self-contained. That is the main reason why DOS approach can work despite its disadvantages.

Scaling DOS is very easy and can be achieved by splitting single collection according to some key. The sharding process is in accordance with map-reduce approach that can traverse multiple machines for each query in parallel.

**Key-Value Stores and Other Simple Storages** This group has its purpose aimed to maximize speed and concurrent access to the data. It can be used for caching or statistics gathering.

Many systems offer multiple value types like arrays or hashes so the DOS approach can be simulated to some level. This may help to store basic objects for an application.

Many systems are able to run in-memory only which further pushes their usage toward non-persistent data scenarios.

Scaling and high-availability is extremely simple to achieve since the sharding key can very well serve the storage key.

**Graph Databases** Graph storage can be relevant in some use cases. These databases have generally very high support for graph-oriented algorithms and that is their major strengths.

Scaling the database across multiple machines can be a challenging task. The graph can be separated by cutting certain nodes and separation regions on separate machines. The benefits are highly dependent on the graph cohesion. If the cohesion is low, the separation is more easily maintained and gives better results.

**Big Table Storages** These storages have been developed in the large clouds to serve a large number of data that is stored in simple schema, for example three-dimensional matrix.

Scaling is their essential functionality.

### 4.3.3 File Storage

A typical application needs both database of some kind for structured data queries and also files that are uploaded or created by application users. Files are stores as-is in byte array form, no other structure exists and they are regarded as a block of data that is transmitted back to user in the same form.

File storage system must cope with the same requirements as other cloud technologies. The main disproportion is the amount of data that is typically transmitted. Single file can be thousand times larger than the structured database record.

Storage systems often have only several actions for files:

- Store file
- Retrieve file
- Delete file
- Rename file
- List files

File locking can be optionally appended to the list of features and makes storage facility architecture even bigger challenge. Directory manipulation may not be supported.

But no other operations are in standard cloud file storage system permitted. No seek operation, stats operation is limited, setting meta-data such as user permissions is often handled in the application itself. File structure is not hierarchical, this can be easily overcome by using certain file name conventions (UNIX-like) etc.

The interface for file storage has been greatly simplified from the one that is available in a standard OS environment. This has a tremendous impact on an implementation and

file storage system is now quite similar to the key-value store, where the value can be of course significantly larger.

This simplified interface does not explicitly dictate to use special purpose service. Already developed distributed file-systems can be also used to fulfill this task.

To ensure redundancy the saved files should be stored at two physical locations at minimum. This feature can be further enhanced by using RAID devices for physical storage to keep multiple copies on the hardware level. There is always a possibility of losing data. It can be infinitely reduced, but will never reach zero. Thus having two copies of single file on two servers in two distant data-centres and using RAID technology on both servers can reduce the possibility of losing the file to minimum.

### 4.3.4 Archive and Backup Storage

It can be defined as file-level incremental write-only storage. The data stored serves only the backup purposes and are never read by the application itself.

The most important feature of backup is the uncorrupted saved data. Even in the case of fatal failures the data must be accessible uncorrupted. The access time can be in the matter of minutes or hours. Often the backup is further copied to external disk drives that are mounted only for the backup process and unmounted for the rest of the day to prevent possible damage.

Only system administrator should access the archive, never the application itself.

The backup area can be significantly larger, often ten times the sum of file and database storage required for the application. The best solution is to backup periodically the whole system with data and then produce much smaller incremental backup (differences against a full backup). This solution is proven by decades.

## 4.4 Price Comparison

General comparison is difficult and is not objective since every type of cloud has a different price model. But to prove one of the main advantages of the cloud (lower price than in a server hosted solution), a generalization to some point must be shown.

Let's image one standard web-application. This standard piece has some infrastructural specifics:

- *SQL storage* for CMS website parts managed by administrators
- *NoSQL storage* for user profiles and user-related data
- *File storage* for user uploaded files content
- *Backups* creation

Every requirement of the cloud application is applied - most notably high availability and distributed infrastructure.

This comparison should be a general guide on how the approach bill is created, now the concrete calculation of current possibilities as this may change in near future significantly. To the general price must include a system maintainer whose salary may vary greatly according to the desired skills.

The comparison is introduced from a medium-large company which is running a public cloud with front-end website and offering clones of web-application software to its customers. The web-application software is open-source and ready to be deployed on the cloud.

### 4.4.1 Own Server Infrastructure

This approach is the oldest one available. Buying the physical servers is of course the highest initial price compared to other solutions. Common servers can be bought for reasonable prices and their warranty is three years. The warranty can be further extended so for an initial investment, the server can properly operate for up to about five years.

Of course this is a physical server that has to run in a server housing company. The HW installation requires company IT worker who can assemble the server, provide HW support etc. The most used warranty type is the next business day solution, so if you have only one server, you can't guarantee high availability at all.

If you have multiple servers, the price goes up, another network equipment like switches etc. has to be bought. Two servers are a minimum for any application requiring any availability guarantee.

For small application set-ups this is quite an investment to make.

Monitoring has to be setup to track server health and availability. In the case of failure, the IT maintainer has to report a failure to the warranty service at least. Or even worse an IT maintainer has to physically visit the server housing company and repair the server on site. Again, with only one employee this is hard to maintain.

This setup has limited scalability options. The virtualization can be used to simulate multiple virtual computers at least.

The application has to be installed from scratch. Every service has to be setup separately, the IT maintainer has to keep everything up-to-date.

### 4.4.2 Dedicated Servers

A lot of companies offer dedicated servers. Basically they buy a server, put it in a server hosting house and completely service it from the HW point of view. HW monitoring may be a part of the service. The initial investment is low, but the price of the servers is spread over months servicing the customer.

This scenario is like the previous one except the HW layer is completely outsourced. If the server failure occurs, the possibilities are quite similar to having your own server cluster.

### 4.4.3 Virtual Private Servers

This approach represents a massive leap forward in the infrastructure development. Owning private servers means owning an anonymous server image that is run in an universal virtualized environment.

HW layer is completely shielded from the virtual server. The major difference to the previous scenario is the fact that virtual server can be easily migrated to any other physical server supporting the same virtualization technology. If the physical server goes down and is beyond any repair, the down-time given by the transition to the new server can be minimalized and in real-world application can be next to few minutes.

Company hosting this solution can very well provide much higher rates of availability that it would provide owning its dedicated servers.

The previous options have very limited scaling possibilities. If the performance is not sufficient, you have to buy another server. In this scenario providers often offer migration between several performace configuration to suit best your application.

The IT maintainer tasks are limited to the server installation only.

The price is dependent on the server configuration that you have ordered.

### 4.4.4 Infrastructure as a Service

This is the first scenario that puts the data in the cloud. IaaS is in a reality very similar to the virtual server setup. Except one major advantage - storage and computation resources are separated.

You can install you application precisely the way it worked on the VPS, but given the fact that storage facilities are generally significantly cheaper to use than general computations, it's reasonable to use the cloud storage.

IT maintainer has to setup only the application environment, most of the services are provided already by the cloud.

Some kinds of non-public inter-server services may be paid. That is in the contrast with your own server installation where there is typically no limit on the traffic between the servers. These limitations are reasonable and are often used to calculate the bill for extra services used such as file or backup storage.

### 4.4.5 Platform as a Service

In these cases the application environment is predefined, cloud users can select one of the preinstalled environments and only additional tuning can be made by IT maintainer.

The pricing models can't be generalized, because they are highly dependent on the infrastructure provider technologies. The separation of storage also applies as in the previous scenario.

### 4.4.6 Software as a Service

This scenario is not applicable to our example. The example company provides SaaS as a result service itself and wouldn't make any sense to use this type of infrastructure.

### 4.4.7 Conclusions to the Framework

The most important point to deduce from cloud pricing is that the virtual machines are cheap compared to other solutions. The framework users will probably use these solutions and it suggests several requirements to the framework itself.

The framework may not be depending on a virtual machine specification as it can change by simply restarting a virtual machine with different number of CPUs etc. The framework must be prepared to detach or attach nodes at any time during its run.

# Chapter 5

# Framework Requirements

After a thorough analysis of the infrastructure, the requirements for a modern cloud framework can be gathered. The framework is a set of tools that is wrapped up in a toolbox for a builder. The builder is the person who decides which tool to use and how he would use it. The framework will support all the essential cloud requirements, but it's always up to the builder if they are used properly.

## 5.1 Goals

There is not a specific area of interest that framework should be targeting, it won't contain any specific business or other application logic. The framework itself will be merely an environment to build an application that supports following concepts:

1. *Simplicity* - every principle should be easy to remember and to be kept in mind.
2. *Ease of Use* - usage of components must be simple.
3. *Distributed* - computation part is easily scalable to multiple nodes.
4. *Fault-tolerant* - a failure on single node must not compromise the whole system.
5. *Self-healing* - after a failure, the system should make steps to recover to certain point if possible.
6. *Programming language agnostic*
7. *Storage system agnostic*
8. *Monitoring support*
9. *Run-time Statistics support*
10. *Minimal maintenance downtime*

11. *Reusable components*

All the requirements strongly suggest that the framework must not be a plain set of API as one knows from other framework projects. It must lead a developer to fulfill a certain set o concepts to achieve all the required features.

A distributed application can span over multiple nodes while each node is self-contained and can operate autonomously. Fault-tolerance means that there is a certain type of load-balancer or failover between at least two same purpose components. Self-healing systems can isolate failure in the node net and adjust the data workflow not to enter failed unit. All these concepts can be achieved on multiple levels.

The network level is one of the most obvious solutions. This has a lot of disadvantages, for instance DNS technology supports this feature, but it can be a great problem because of the randomness included. Any change to the DNS record is propagated slowly, so it's virtually impossible to load-balance by this feature. The request amount always fluctuates in the cloud. It's a problem to have a static infrastructure in the cloud.

The main conclusion is that framework should offer a self-contained computation unit, let's call it a module. This module should have an interface, should process messages and return certain messages to the sender. The module serves a certain fixed purpose, can be started or restarted at any time or even started multiple times to load-balance a pool of messages.

Agnosticism to a programming language and storage facility is an essential concept. The disadvantages of bindings to some kind of proprietary technology have proven devastating to many projects in the past. The need to change a storage facility may very well occur several times during the application lifecycle.

The monitoring and statistics management is a necessity in a complex long-term running application. Current projects also need to maintain high availability so it's not possible to take the whole cloud down for update etc.

The requirements will make the framework quite complex, but I believe that the framework layer is the most flexible part where these problems need to be addressed. The innovation of such a framework will mainly lie in its concepts and goals, not in pieces of source code.

## 5.2 Modern Framework Guidelines

There is a lesson to be learnt from many frameworks that already exist. Modern approaches not only support traditional application development, but also decrease development time and increase product quality.

Following concepts have been selected on purpose because I personally believe that currently each of this technique can produce a state-of-art application in a certain field.

These techniques combined should provide solid theoretical background and guidelines for upcoming framework.

The guidelines should be coherent as much as possible and form a synergy together. The resulting product should have at least some level of quality at each of the discussed domains. The effect should be a large quality improvement over existing applications.

**Test Driven Development (TDD)** TDD is considered a standard for any application currently developed. Our main goal is to test modules that have certain interface and on certain input outputs exactly one result value. This approach is called black-box testing because the exact functionality inside the „box“ is not known to us [25].

Unit testing is another time-proven feature that is considered essential in maintaining product quality.

Tests must be written before the implementation part begins. Then the implementation has the only role - to pass all the tests. When it does the work is finished and a next iteration can start. This sequence is sometimes hard to maintain in the real world, some amount of code must be prepared only to run tests correctly, a large amount of behavior of application is testable (accurately measurable).

**Domain Driven Design (DDD)** This approach dictates among other things to create a thin bottom layer of domain objects that interact with each other. By their interaction the application is driven and responds to their states [1].

The creation of so-called ubiquitous language is the main advantage of this methodology. The communication in the development team, between the team and domain expert is held in this special kind of well-defined and technologically precise language.

The domain behavior is then just processes description in ubiquitous language.

This principle is one of the most essential. The framework could exist without its support, but that would most certainly mean that a large amount of developers will consider using standard CRUD model instead. The CRUD model is strongly discouraged as it is extremely difficult to maintain in enterprise level environments.

**Behavior Driven Design (BDD)** BDD can be seen as an evolution of the TDD together with the DDD technique. The concept is to test objects and their behavior. This approach is excellent for domain objects.

**Mocking** Mocking addresses testing issues in large systems. Mocking allows a creation of dependency with a fake dependency.

With such dependencies the parts of a system can be tested in separation.

**Command Query Responsibility Separation (CQRS)** Complex application have a lot of logic built around a object or document storage. A lot of effort is paid to keep saving the object state similar to reading its state.

But in a reality this is often redundant and pointless. The CQRS patterns suggest to separate command layer (writes) and query layer (reads) completely.

This and nothing more is the suggestion by the pattern itself. Many more approaches exist as a consequence to CQRS. Their main ideas evolve the concept further, for instance to have completely separated logic for storage (for example complex ORM) and unrelated for data retrieval (for example simple SQL query).

**Event Sourcing (ES)** Every change to the domain model can be seen as an event. The domain objects interact with each other producing events that change their inner state. Events can be recorded in event store and further analyzed. With events in the store it's possible to reconstruct a system to any moment in time without any database-level backups or other technologies.

ES is a perfect choice for integration with CQRS. The command issues the change to the domain objects. They produce events that are immediately applied and stored at the same time.

**Dependency Injection (DI)** The components inside the framework have dependencies. The framework should provide means to meet these dependencies semi-automatically. Each object should have all its dependencies setup during its creation. This is called constructor dependency injection.

The created object is always in a working condition, it may not be created if one of its dependencies is not working properly.

**Non-intrusive Framework Integration** One framework should be replaceable for other with similar capabilities. In practice the framework should provide interfaces or other loose binding for its features.

The framework must not dictate abstract base classes that have to be inherited or other approaches that would result in high coupling with the framework classes if there is a loosely coupled option present.

The approach must be coherent throughout the framework and offered options. The framework design in this area is simpler these days than before because a lot of language-level specifications are implemented through pieces of meta-information inserted to regular source code. Pieces of meta-information are collected on the first run of the system and the processing or service registration is made by these so called collectors.

**Domain Specific Language (DSL)** DSL has been given recently a great attention with the rise of new programming languages. DSL can represent virtually any custom language that is enhanced to describe a target domain better and simpler than traditional programming methods. The builder enterprise pattern often stands behind as a custom DSL implementation.

Using DSL is much simpler for target audience and many tasks can be achieved by simply „scripting“ in DSL.

**Representational State Transfer (REST)** This idea is a sort of resurrection of the original models that formed HTTP. Request methods have been restricted to only GET or POST, others have been ignored. The REST or the RESTful movement suggests to use other methods with slightly altered meaning again.

REST approach is becoming more and more standard in intersystem APIs.

**General Responsibility Assignment Software Patterns (GRASP)** Most notably loose coupling and high cohesion are the most important enterprise patterns used when designing a framework. They provide basic blocks for scaling framework horizontally and concentrating the logic to short dependency circuit of classes.

These techniques are not inovative and do not push developers to new way of working, but tend to improve the quality of the old and rusty projects.

**SOLID principles** The so called „first five principles“ are essential rules for building quality application code. They support ideas of incremental refactoring, to increase cleanliness of codebase.

**Multi-Tier Architecture** Clean separation of logic into several architectonic layers is essential in building robust and scalable software. One layer is responsible for maintaining the database connection and communication channels, other is composing objects and maps them to the storage.

**Usability in Enterprise Applications** Enterprise applications are a typical example of a large software ecosystem that can be incrementally developed for many years or even decades.

In such a system there is always a large number of components and frameworks in cooperation at any given moment. The framework should recognize its sovereignty and not push other frameworks out of the system by certain type of artificial limitations and so on.

**Don't Repeat Yourself (DRY) Principle** The aim of this principle is quite clear. One function should be contained in only one location in the source code. No logic should be duplicated as long as it has the same function.

## 5.3 Required Services

The goals of the framework are given. There are two separate dimensions that the framework helps to accomplish.

The framework only provides the means, to choose specific technologies is up to a developer.

### 5.3.1 Core Framework

These are essential needs to support further framework extension. Without this core or microkernel the framework could not work. General features are declared in this section.

A lot of these features exists in core package only as an abstraction or interface with no fixed implementation. The implementation is provided in Integration package as a specific integration with some kind of technology.

**Multi-threading** Threads are essential in building scalable application. There are many levels of multi-threading support ranging from classical C language approach (Dining Philosophers Problems etc. [26]) to high-level point of view in Erlang language (Actor-based concurrency [27]).

This field is well known with its traditional hard-to-find bugs and non-reproducibility of tested situation. In my opinion the higher level approach is used, the better the results are. Developers are not machines to carefully analyze every possible scenario of thread execution plan, to enumerate all possible values for each memory cell to conclude that the concurrency is safe and no problems may arise.

This situation is not going to happen. If the concurrency is tackled at low level, deadlocks are inevitable. A large amount of applications acknowledge this fact and build complex runtime deadlock checkers that can detect this situation and recover from them.

**Messaging Infrastructure** It's inevitable to separate application to several self-contained building blocks when developing a large application. Transparent messaging interface is the way to go for inter-blocks communication.

The messaging service must be as stable as possible. It's always a messaging layer that provide fault-tolerance up to some level - delaying messages until the recipient comes online etc.

The service has to span across several virtual machines to provide required key functionality elements.

**Remote Procedure Call (RPC)** RPC is a well-proven standard tool for inter-process communication. For certain types of work on request / response model, the RPC technique is the best solution.

**Events-driven Architecture (Publish / Subscribe Pattern)** Another type of service is event publishing. An asynchronous basis of events allows multiple responders to register for single event key. As the event is fired, every responder must be notified.

**Event Store** Events are the heart of event sourcing architecture. An event stream records every change made in the system. Events must be stored in the event store. The events can be represented as highly structured immutable objects. Every event is attached to an aggregate root which is a central concept in DDD.

The event store must support methods for:

1. listing aggregate roots
2. retrieval of all events for certain aggregate
3. creation of new aggregate roots
4. appending events to the aggregate root (atomic operation)

**Projection Support** Projections are key concepts in CQRS and DDD techniques. Their main objective is to prepare separate read projections for each read-level scenario. Projections are created as a reaction to events.

In fact, every projection can be dropped and reconstructed to the same content from an event stream at any time.

Projections are often created in SQL and NoSQL databases, because these technologies allow structured and fast data retrieval.

**Continuous Integration Support** One of the essential aspects of the framework must be a wide support for automated testing and continuous integration (CI) in general. The effort is expressed in supporting tools for automated building assembling the framework project.

### 5.3.2 Database Support

Both SQL and NoSQL databases are well-proven and mature enough to be used in this framework.

These databases can be used as an event store. They are also valid for created projections.

### 5.3.3 File Storage Support

The file storage service is an extension of a simple disk-file storage provided by traditional OS.

The service has a simple interface and except for stability and availability requirements there is an only differing issue concerning data throughput that is far greater.

### 5.3.4 Monitoring and Statistics Collection Support

The support should be built-in to the framework itself up to some pragmatic level. The monitoring is essential for a long-running application. In simple cases the basic metering like allocated vs. used memory, enough memory or disk space should be enough.

# Chapter 6

# Framework Design

In my opinion the best approach to start the design process is to focus the attention to a simple model that is iteratively enhanced. As technologies and principles will be added to the concept the framework will become more and more robust.

## 6.1 Basic Concepts

The root of the framework is an actor-based concurrency environment. An actor is a simple concurrency entity. An actor has a mailbox represented by a message queue. When a message arrives, an actor is activated and reacts on a message. An actor can send messages to other actors.

The idea is quite simple at first - a launcher element will create each application component in parallel. Each component is a self-contained service that has certain dependencies and offers certain services.

Here comes the first necessary technology we need to provide - dependency injection. This is usually provided by a container that keeps track of all the running components and is able to launch a new instance of certain component type with required resources.

Let's call the component a module. This is a more traditional label for the same feature. Module is of course self-contained, can run any number of actors on its own without letting the top-level container know.

**Messaging** The application is composed of several loosely coupled modules which can interact with themselves. The amount of communication can differ significantly in various use cases and there is no general rule how large a module should be in the matter of memory or computational power. Clearly any module can be a bottleneck and may prove to slow down the application. The framework should help and allow a developer to deploy the same application in different deployment scenarios where each node runs a different subset of modules.

The trivial use case is when all modules run on the same machine. After performance analysis, slow modules can be moved to other computational nodes with minimal changes to the application code. Every inter-module call should be handled as a message that can be transparently serialized in case when module is running on a different machine or simply passed over if the module runs on the same box.

Message passing is the key concept in the framework and is used to achieve a number of design goals. The message handling should be as light-weight as possible to minimize any overhead in module communication.

**Dependency Injection Container** Dependency injection container has to register the components in certain order. The requirement order must be linear and any dependency cycles are forbidden.

The DI container creates an environment composed of components. In traditional applications the number of components inside the container is fixed and it is not possible to add more component types once the container is started. This simplifies a programming logic a lot as there are dozens of use cases that can occur if the components are allowed to register more components.

The features that the DI container has to offer are:

1. *An initial environment configuration* together with a bootstrap (components started at boot time)

2. *Available components* - repository with all the available component types that is able to launch new component instances

3. *The launch list* - list of components with configuration that are scheduled to start. List is initially filled by a configuration, more targets can be added by components themselves or primary node command later on.

**Error Recovery** Errors always happen. It's a completely inevitable truth. A lot of developers try to analyze the code deeply and pinpoint every possible exception being thrown. It's impossible to be prepared for every existing failure scenario.

Much better strategy is using supervised class hierarchies. Since every actor has a simple lifecycle, it can be shutdown in a case of failure. When a failure happens and the actor can't handle it by itself, it delegates it to its parent. The parent can decide what to do. One obvious solution is to take no action. This is improper in many scenarios as the failures leaves the actor in faulty state generating more errors as the application continues. A better approach is to stop actor and let it start fresh with all its children and inner infrastructure.

Starting the system is commonly much better tested feature that letting the system work when some part crashes.

## 6.2 Modules

The module must expose an interface of some kind. This is an issue that has to be considered carefully. A module can exist multiple times in an environment. If the module is updated, the new version of the module can run in parallel to the previous one in the same environment. There can be changes to number of optional parameters or so on and the new module must handle the old version message.

**Serialization and Deserialization** The solution is well-known of serialization and deserialization procedures. An interface must be declared as an immutable object with fields. Some fields are required while others are not. The key to keep object compatible with any newer version is to never add a required field to the object. If such a field must be added than a default value must be set for it. The result is that each version of the „interface“ immutable object is easily convertible to any other version of such class.

Serialization is an important topic when designing cloud systems because its performance has serious impact on the cloud itself. A serialization should be simple to use at the same time. This may seem like a contradiction to the current technologies.

The solution is to include content-type header in messages determining the real message format. Services can choose the serialization method along with input DTO for each provided service. They may even listen to several message formats at once and decide how to act according to the content-type header.

**Routers and Load Balancers** Modules can contain their own service hierarchies. But it is possible for a module to encapsulate another module and control its function.

If a module starts a sub-module, it has complete responsibility over its actions. This behavior can be positive as the module can mimic network components.

A module can provide a simple fail-over or a round-robin forwarding to other module.

As each module can be executed on a separate node, this concept is essential for large and scalable applications.

**Message Traversal** Developers tend to use remoting and interaction with remote computers as a simple „remote procedure call“. But if the messaging is used extensively there is always a problem of resources that are allocated to wait until the message arrives back and normal program flow is resumed.

This approach is likely to consume a large number of threads and a lot of synchronization issues may arise as the return value from remote source return in nondeterministic manner.

A significantly better approach would be for each process to create a pipeline where no RPC and synchronous waiting for the reply are used at all. The main difference is not using the ask operator on actors but instead using the result message type as the one

listened to for the sending actor. No threads are wasted as after the request is sent the procedure ends and the thread is returned to the thread pool. When the message arrives it is matched as any other type of received message by the sending actor.

This approach converts synchronous communication to asynchronous which is better suited for actor systems.

**Module Interface** Modules provide services. Each service is defined by:

1. a relative or absolute address in an environment that is used as a referral to such a service.
2. input interface (DTO) - similar to function arguments.
3. output interface (DTO) - similar to function return type.

DTO pattern is used to exchange data. This model guarantees immutable object states. This is extremely important, because in a multi-threaded environment, the invariability of the arguments is welcomed as it brings more stability and security to developers.

The DI container has a configuration of target modules to start. The modules requirements are iteratively collected, a linear queue is created and then started up in parallel.

For a module configuration a DSL should be created to minimalize the potential errors.

## 6.3 Command / Event Pipeline

The framework does not offer only simple request / response principle. It must support holding the internal application state. Several framework requirements cover this area.

CQRS suggests that the write and read logic should be separated. The separation is done by separating commands (write requests) and queries (reads). Commands are send to Command Handler and domain objects are altered in reaction. Queries are run on the projections. A projection is an output mechanism that processes events and alters an output table or collection.

The key feature in simplification the domain logic handled by the framework is to use the DDD technique. Of course at this abstract design level, there is no application logic yet. But the support for DDD can be prepared by designing the command-event a pipeline.

The most important element is an aggregate root. These aggregate roots are created from event stream. The process can be demonstrated by pipeline:

1. A command issued
2. Command Handler Bus processes the command

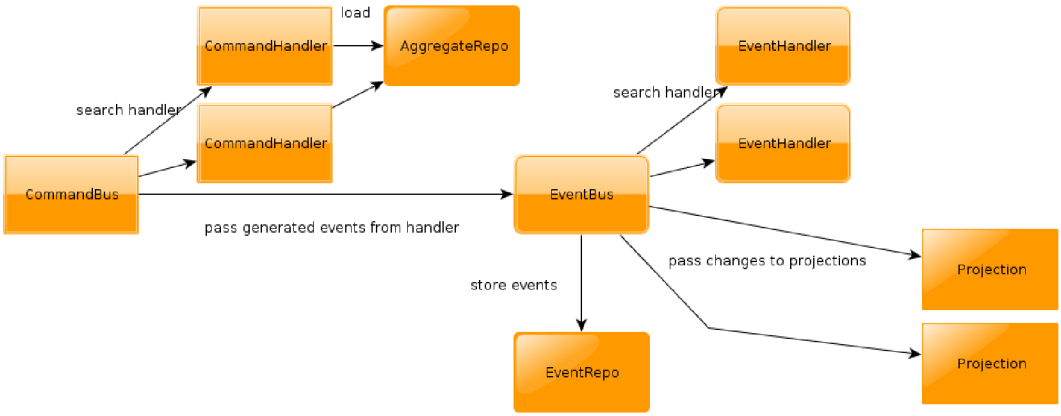



**Figure 6.1:** Pipeline Overview

3. Specific Command Handler (CH) operates on an aggregate (or creates one)
4. In case of failure the CH returns a failure
5. An aggregate root has an event stream attached (generated by domain action in CH)
6. All generated events are atomically saved to event store
7. The CH returns success to the caller

Aggregate roots are hearts of the domain logic. They have method in ubiquitous language that change their inner state and generate events with equal function as those changes.

The event stream is then pushed to projections. Projections are responsible for listening to certain events and updating the database parts of the system. The process can be modeled with few simple criteria:

1. Events are read from Event Source
2. Events are pushed to all listening projections
3. A projection filters only important events
4. Filtered events are applied to the DB

It's important to mention that error can occur only when processing the command. Therefore its result must be returned. Once the events are commited to the store, the projecting them must not fail. Of course in case of an exception during the projection process, the whole projection can be discarded and recreated once again from the repaired projection process.

## 6.4 Testing

Testing is an essential part of every application that must be integrated at framework level.

**Testing Components** Unit testing the components is pretty straigh-forward. A special test-actor having an ability to trap incoming messages is used. The environment is started in some configuration and afterwards tested actors are replaced with this trap actor.

**Testing Domain** Using DDD guarantees fairly simple testing. From the BDD point of view, the unit test of the domain means simply creating an aggregate object in some state and calling its methods. Each method call represents a command being executed. Each changing command generates one or more events.

The target of the test is to test if the generated events explicitly equal to some predefined list.

The domain is the lowest level of the application, every connector etc. is above this layer. Every tested service can be replaced with trap implementation just to ensure that the call has arrived to the method.

## 6.5 Statistics Collection

In a long running application, runtime statistics are needed to be watched closely. They provide accurate information about the application usage and free capabilities. With usage of complex virtual machine and execution environment it is difficult to measure resources consumption at process level. The calculated number vary depending on many unpredictable variables such as garbage collection interval.

There is no point in measuring everything in the application. Real or possible bottlenecks should be analyzed and these metrics need to be measured.

The main problem with the statistics collection is the extra level of CPU usage it can generate. Developers must choose carefully how often to collect pieces of information and which of them needs to be stored.

The statistics architecture is therefore declared as a set of callback interfaces provided optionally by each module. The collector interface is passed to each module on creation and each module registers statistics counter. These are simple routines that return some kind of output.

Statistics are collected from a central point by external application only. The external application can be tuned at any time to collect more or less often. It can be turned off completely if the application load rises beyond some critical level.

## 6.6 Self-healing and Self-testing

More specific healing mechanism must exist apart from simple supervision control that restart part of the actor system based on exceptions. Health-control system should allow each module to validate its state, required resources and it's up to module to choose an appropriate action.

A module should react to a call - self-diagnostics. This special message has its only purpose - to allow module to analyze and separate maintenance mode from standard requests.

Diagnostics can result in a suggested action - the module can count its request count and if it reaches some level, the diagnostics message can result in a request to add more workers to the pool etc. If no action should be taken, the module at least announces the interval to another check.

If the module does not reply in a specified timeout, it is considered faulty and is restarted or replaced.

The self-healing mechanism has main purpose to allow self-healing and adjustments to module. For example the module depends on a projection that is generated with errors, then the module can detect this fact upon diagnostic message and start the projection reconstruction.

## 6.7 Nodes in the Cluster Structure

Cloud structure has to fulfill the requirements for high-availability and fault-tolerance. Complex design decisions have to be made to offer these features at any application state.

The framework is suitable to the cloud environment of virtual machine. Apart from its dependencies to other daemons (storage, db etc.) the application should be one process that creates and maintain single node-level environment. This is the environment controlled by single operation unit that provides following services:

1. launches configured modules
2. maintains node DI hierarchy
3. DI discovery for any module
4. local health checks
5. message interface connectivity
6. exposes special interface for alive check
7. maintains node state

The main benefit of the environment is an ability to launch multiple nodes in parallel. There is no leader node present. Each node can operate autonomously only on the status information given by other nodes. Of course this raises some security issues in the future.

### 6.7.1 Primary and Secondary Nodes

This approach may represent a different model of leadership for application with security standards or where there is a need for centralized management of the components. This approach could be mixed up with no-leader model to create a hybrid leader system.

For cluster managements a responsible operator can be chosen in order to make any adjustment to the cloud. There are many types of control decision mechamisms, but many of them suffer when it comes to even number of nodes. The resolution is not possible in these situations. Modern approaches suggest adding „blank“ nodes to the cluster that has the ability to vote only. These are configured by administrator up to his preference and a knowledge of the virtual server installation.

The rules for selecting primary node are simplified rules from these modern approaches. The rule is quite simple actually - every node has a knowledge of following:

1. Unique priority numbers - the larger number decides the primary node
2. List of all nodes with their priority number
3. List of reachable nodes with their priority number

On any change in the cluster (network breakdown, new node addition) a following procedure is held:

1. The node state is set to secondary
2. List of reachable nodes is refreshed
3. If the number of reachable nodes is higher than half of the total nodes continue, otherwise the cluster part won't have any primary and must wait for rest of the cluster to reconnect or be restarted. This is considered a severe incident and should be handled manually.
4. The node with the highest priority number becomes primary and notifies other
5. Refresh status of all the dependencies of the running modules on the node, request their start on the primary if they are down

This algorithm is simple and sufficient to our scenario because primary node doesn't have many responsibilities and selecting machine with low performance is not an issue.

The primary node has following privileges:

1. Move primary token to another node
2. Fail-arbiter - if the node with certain service required by other running module goes down, the fail-arbiter decides where to start a new instance of a service.
3. Main statistics collector - this node collects every node statistics
4. Node monitor - alive-checks other nodes

## 6.8 Updating the Application in the Cluster

If the best practices are kept, each node should be self-contained. If node goes down, every required service is started by primary node at some other node. It should be no trouble to bring down any node if there is still redundant capacity.

The update of the node is quite vague term. It can mean updating only the application and restarting the node or it can mean replacing some HW part of the physical server. In either case after the update the node is brought up again and started according to the configuration. It is recognized by the current primary node and attached to the cloud.

## 6.9 Heterogeneous Nodes

The minimal defined interface for the node is minimalistic - only the SystemNode with required abilities like providing dependencies and starting modules. The module internal composition or even the node internal composition is not enforced in any particular way.

The node should provide modules in order to be of any real use, but even this is not a technical requirement. Special node types could be developed for node with high priority number that is customized to handle statistics collection well.

## 6.10 Debugging and Error Diagnostics

The debugging is a challenging task in a complex system. With multiple nodes running across multiple virtual servers this becomes even bigger problem. One trade-off of avoiding RPC messaging scheme is that the exception stays contained on the called module. This may not be a desired behavior since the caller does not implicitly know that something went wrong and an exception was thrown.

Every exception in the system that is not handled must be logged and it's up to the system administrator to take action. In the complex system a lot of exceptions can raise false alarms since the connection errors occur or some timeouts are reached. All these events should be easily recoverable by restarting a part of the actor tree. This applies only if the recovery is possible of course (network connection is restored etc.)

In the future it may be helpful to send a report of each exception to the administrator or to count the number of errors in the statistics to signal some kind of problem.

## 6.11 Framework Composition

The framework itself should be divided into several areas that can be implemented as loosely-coupled packages:

1. core - This package contains all the necessary interfaces. No other implementations are given. Implementation of the interfaces is up to the custom extensions.
   (a) messaging - Serialization Mechanism and Other Messaging Features
   (b) module - ProducerModule
   (c) node - System Node
   (d) command - Command Handler Support
   (e) event - Events Support
   (f) projection - Projections Support
2. extension - Implementations of the core interfaces in specific technology
   (a) amqp - Messaging AMQP support
   (b) http - Basic HTTP MVC components
   (c) mongo - NoSQL storage
   (d) and other technologies

The extension package is as the name suggests a subject to further extension when a new and more progressive technology appears.

## 6.12 Building an Application on the Framework

The application is built on the framework. The framework is launched with configuration options. Once it is launched, it creates its own thread pool and every other required resource. It runs in complete separation and can be controlled through a simple interface.

The application in the DDD style should start as following:

1. Form a ubiquitous language etc.
2. Build the domain objects with events emitters
3. Build command handlers and register them to the command handler bus

4. Design and implement EventHandlers

5. Build desired projections and register projection readers

Apart from this DDD programming style support the application can expose modules to the DI containers. The module is registered simply by offered services.

The framework will cover a lot of boiler-plate present in the application and plumbing between components. It will provide advantages like scalability, high availability and much more if used correctly.

But a lot of applications will need a user interface attached, let's consider a web user interface. The industry standard for building web application is the Model-View-Controller pattern. The model part is the DDD domain object layer, the controllers are the entry points that receive requests. Such controllers could be designed as modules to suit the model well. The view components could also be modules with simple interface - taking the DTO with every object required in the view template and generating result output from some kind of a template.

The only component is needed - the HTTP server interface router that opens a port, receives and parses incoming traffic and forwarding the parsed request to appropriate controller (e.g. module). The requests are synchronous, the caller is very likely to use a blocking RPC approach.

As the application development progresses, it should become more robust adding more module types available to the DI container. Also as the applications' need for speed or stability increases it is possible to extend any of the implementation of key components (DI container, Operator Unit etc.).

## 6.13 Moving away from CRUD to DDD

There is a large gap in the amount of information on the traditional approach for designing information systems - the CRUD (create/read/update/delete) style. This style is directly linked to the SQL database storage system. The application is simply a „controller“ for the database tables, each view handles one particular table, allowing user to perform basic table operations. This development style is well-known, easy to implement, but has a major disadvantage - the application usage is tightly-coupled with the database structure. The user has to understand and learn the database structure in order to operate the system.

The modern methodologies like DDD suggest using use-cases as the basic units of operation in the application. Every view is essentially one use-case allowing user to make only one certain operation.

The CRUD approach can be beneficial in very simple applications, as the complexity grows and every operation is more complicated, the CRUD model fails to comprehend the

complex scenarios. Every complex consistency check on the database is implemented like a hook for update/insert/delete operations. The DDD on the other hand adjusts each view according to the use-case with all its loose ends and in most cases allows developing with linear complexity.

The DDD approach is better suited for much more real-world applications in my opinion.

# Chapter 7

# Technology Selection

The used technology is essential to the framework day-to-day function. Mature and stable technologies must be used in order to let the framework inherit these properties. The technologies evolve and the framework should be prepared to adopt new databases or even programming languages. The framework should support integrating new technologies at some level even if they are binary incompatible with current codebase, it should be possible to use new module through messaging etc.

## 7.1 Technology Evaluation

It is not possible to evaluate all the available technologies that enterprise market has to offer. As one of the framework requirements was the programming language agnostics, it can't be pinpointed to one selected language and conclude that it is the best possible solution.

There are most certainly thousands possible solutions and many of them could work correctly in a real-world situation respecting all requirements and fulfilling all framework goals at the same time. It's far beyond the scope of this thesis to evaluate or even select across multiple candidates.

I have selected several enterprise-grade, well-known and time-proven technologies mostly because of my own experiences and preferences.

### 7.1.1 Java Virtual Machine - Runtime Environment

JVM has been developed for decades and its known as stable enterprise environment for byte-code execution. JVM provides a large amount of features from an advanced garbage collection (GC), real threading (no emulation), HotSpot optimization technology. The main disadvantages are:

1. execution speed - this is not true anymore, since the HotSpot technology can optimize the code to be almost as fast as the native compiler would produce. There

is little overhead compared to native languages, but the difference is minimal on current hardware.

2. JVM startup time - the environment has to boot before the application is launched. There are mechanisms to reduce the latency, it is no longer a real issue for such a large number of applications

3. Memory limits can't be changed during runtime. This is an issue, because the JVM process has to start with declared amounts of maximum heap and perm-gen space. These limits are fixed for an application lifetime. The best solution to this issue is to monitor the application during runtime and then provide it with reasonable large amount of memory. The statistics in the framework can be used for gathering memory usage.

### 7.1.2 Scala - Native Framework Language

Scala has been chosen for its ability to run on Java Virtual Machine platform and for its language features. Scala means SCAlable LAnaguage and can be easily extended to virtually any form of DSL that is required for the job.

Scala provides real threading capabilities (on top of JVM) which are also essential to the application.

### 7.1.3 Akka - Actor-Based Concurrency Model

Akka is built on top of Scala infrastructure and provide concurrency at actor level. Akka has a large toolset for building highly scalable applications.

The remoting support to connect event actors among several computers makes it an ideal candidate for framework substructure.

### 7.1.4 AMQP - Messaging Infrastructure

AMQP provides communication over multiple entry point that is easy to use. The communication infrastructure is composed of multiple endpoints called exchanges, on which clients listen and receive messages.

RPC communication can be easily implemented using temporary queues.

### 7.1.5 MongoDB - NoSQL Database

MongoDB is reliable document storage facility. It has been selected as a general representative of the no-sql database movement. It is quite sophisticated compared to other databases in the group, but it is easy to use and the main query and command mechanism is the JavaScript language.

### 7.1.6 PostgreSQL - Possible SQL Database

PostgreSQL is currently the most advanced open source database. It is used by several cloud providers for its high performance output.

### 7.1.7 Neo4J - Graph Database

Neo4J claims to be the most advanced graph storage engine available in an open-source environment.

### 7.1.8 GridFS - File Storage

GridFS is file-system built on top of MongoDB and provides advanced file system functionality.

The GridFS client implementation is easy to integrate to a new framework.

## 7.2 Node Daemons Configuration

There must be a specific environment set up to work on selected technologies. The cluster environment is supported with every selected technology up to some point. Some suggestions must be made because any daemon installed on a single node without replication could bring down the entire data storage on hardware failure.

### 7.2.1 IaaS Provider

The environment is built on the IaaS infrastructure which must be guaranteed to be reliable. This can be achieved only on the IaaS provider side by using quality hardware, having simultaneous connections to multiple providers of electric energy and internet connectivity. These metrics are individual and there are no general comparison rules which one is better or worse.

### 7.2.2 RabbitMQ - AMQP Messaging Endpoint

RabbitMQ is popular AMQP routing and messaging server. The main advantages are extremely high stability and support to scale messaging nodes.

The cluster works by replicating all the settings and internal configuration across all the servers. The application driver for AMQP has to accept multiple servers in configuration. The driver should reconnect to other node if the current node fails.

The nodes are equal and should be identical most of the time. In the worst case scenario, an extra transmission over network is generated when applications connect to different RabbitMQ nodes.

In the case of failure the RabbitMQ cluster is able to semi-automatically repair itself. After the reconnection of failed nodes the synchronization is launched and node is reconfigured in the matter of seconds.

### 7.2.3 PostgreSQL Cluster Setup

The PostgreSQL can be launched in several ways in order to achieve a simulated high-availability. The database system itself supports only simple replication in master-slave pattern. This could be a large problem in case of a failure. There are certain third-party options that suggest placing a forwarding component in front of the server itself. Depending on the vendor, the component can provide load-balancing, fail-over or simple connection pool. All these options have major disadvantages that are not considered a standard in the PostgreSQL community.

The main problems in the master-slave concept are:

1. Master / Slave is determined in configuration and must be set before the application is launched. The application must be at least restarted in order to switch from Slave to Master.

2. Complete resynchronization is not automatic. In these cases Master node must be set to state when it writes to the memory only, the files stay intact. The files are manually copied to Slave node, the Primary is set to standard state and after starting the Slave the replication may begin to work again. This approach is nontrivial and extremely hard to automate since there are many possible scenarios of failure and only some of them require this resynchronization.

3. Asynchronous replication is the only possible solution for larger clusters. There are no advanced strategies like a quorum or others that would help to decrease latency and data security in larger installations.

These problems are not fatal and given the probability of a failure should not forbid the PostgreSQL usage in the cluster. These are most certainly serious potential problems that may degrade the database function in the case of a failure. There are two possible scenarios:

1. Slave failure - should not pose a serious problem as there should be multiple slaves and the application can just switch to another one.

2. Master failure - depending on the severity of the failure, the solution to the problem may vary from restarting the daemon process to failing over to „warm standby“ [28] as referred in server documentation. The failover is not a simple step and there are many possible scenarios when the failover should not occur and in consequence this

could lead to data loss or data corruption. The Master failure inevitably requires some level of administrator manual interference and can't be automated.

These problems are common to SQL storage systems and the possible solutions are limited, these technologies are simply not well established in the fields of a high-availability. They are of course usable, given the fact that reading is less likely to fail, they are a perfect match for projections. A projection is generated from an event stream and by definition can't contain wrong or inconsistent data at any time. If the failure occurs, the data will be outdated depending on the length of outage, but there is nothing more to it.

### 7.2.4 MongoDB Cluster Setup

MongoDB has been known to serve the cluster well. There are many replication strategies and supported use-cases even for multiple interconnected clusters.

The main-stream approach is to use Primary and Secondary nodes, where the Primary is the only node with write access. In the case of failure a new Primary node is elected by a sophisticated yet simple process. The application is configured to use multiple nodes at the same time and on a request can be informed which node is the Primary.

The node synchronization is completely automatic after the failure or other data-losing event.

### 7.2.5 JVM Usage

Depending on the use-case there is a possibility to launch multiple application nodes on a single virtual server. This can make sense in a lot of scenarios where a periodic action is taken and special node is attached. For most installations there should be only single application node on one virtual server. This approach leads to simple memory and thread pool management.

The process itself is unlikely to fail as the framework is able to catch most of the application errors. However there are cases in which the application causes segfault or exhausts the memory. Therefore there is a recommendation to launch an application under some kind of supervising server that is able to restart it automatically in the case of a fatal failure.

## 7.3 Other Suggested Technology Stacks

The technologies selected for a demo application are well known and have proven during years in many business-critical applications. However, there is always a freedom of choice of the technologies as the framework connects loosely-coupled components through transparent messaging.

There is no need to stick to the same technologies during multiple stages of development of the product. It is quite natural to adopt new languages or databases as they evolve and join them inside the technology stack.

There is always a possibility of replacing the entire module or even a node if other implementation performs tasks better.

# Chapter 8

# Framework Reference Implementation

The framework is implemented together with the demo application. The best source of low-level documentation available is in the source code. The high-level implementation details are explained as there may be multiple strategies to approach problems.

## 8.1 Implementation Choices

**Serialization and Deserialization** For clarity and demonstrational purposes, the selected method of serialization is JSON format. This format is human-readable and can explain principles far better than other byte-oriented format. The serialization to Scala native value types is direct and in most cases requires no further explanation.

The only possible drawback is missing validation mechanism for the format itself. This problem is easily addressed with Scala case classes - these classes are DTOs which declare property fields. The field may be assigned a default value. When the value is not provided, the serialization process must set a value or an exception is thrown otherwise. There are also possibilities to use the Option class.

In a real-world application, the exchange format could stay JSON or if the speed would be an issue, more performance oriented protocol such as BSON or Protocol Buffers can be used.

Only case classes should be serialized or deserialized. This is a general rule and a Scala standard.

**Module Interface** The most critical part of the implementation is the ProducerModule interface (or in Scala: trait or abstract class if a constructor functionality is required). The interface should be simplistic as possible fulfilling only the required functionality. It is declared as a standard base class:

```
abstract class ProducerModule(name: String) \
        extends ProducerProxy with ModuleRef with SLF4JLogging {

        def handle: PartialFunction[AnyRef, Unit]

        def receive = {
                case other: AnyRef =>
                        if (handle.isDefinedAt(other)) {
                                handle(other)
                        } else {
                                log.debug("Message not handled: " \
                                        + other.toString)
                        }
        }
}
```

The module is a thin container for actors. The crucial element is the handle method to which is delegated to the actor call. It is an actor itself. This is necessary for the modules to be able to register services on the same level. The services are interchangeable and there is no need to know the correct node if addressing other modules by a unique name. For addressing, there is a ModuleRef interface:

```
object ModuleRef {
        val ModulePathPrefix = "/user/supervisor/"
}

trait ModuleRef extends Actor {

        def moduleRef(name: String): ActorRef = {
                context.actorFor(ModuleRef.ModulePathPrefix + name)
        }
}
```

**Module Lifecycle** The module has a defined lifecycle. Its methods are called back on certain events and can take appropriate actions. Standard actor lifecycle events are also used.

1. StartNewModule - when the module is started by the DI container
2. Forward - used to forward messages through the remote actor

3. StopSilentlyModule - before the DI container is shutting down the module

For each service the lifecycle is important. The order in which the services are started may not be fixed for each module. A ProducerModule instance can be created with statistics collector to ease the statistic types registration.

Each ModuleService is an actor, so same lifecycle applies for them too. Service can declare its own supervision strategy to prevent further exception escalation.

Service receives the requests in the deserialized form (given by the deserialization function).

**Routing and Failover ModuleService** One module can contain multiple actors inside, but is completely responsible for their attendance. The parent module should create a ModuleRef that is able to route or forward message to child services. The parent module can register child module services as its own children except a simple name-prefix in their registration path. The prefix makes these services effectively private since no other module knows the prefix.

Forwarding is desired in many scenarios and it is quite common that apart from standard requests for the target service there are special control requests that are handled by the parent service itself. For example: a request to start another child or to kill a random child.

## 8.2 Implemented Services

**Dependency Injection Container** DI container has few major responsibilities:

1. Start Module, and maintain their lifecycle.
2. Maintain local list of currently running ProducerModule and checking their status.
3. Maintain list of available types of modules ready to start if needed (dependency crash on other node)
4. Communicate with other nodes to determine if the requirements for a module start are met.

The ProducerModules are started as actors named after the path that the service is providing. This simplifies the address discovery for services.

**System Node** The system node is the launcher element of the application node. Apart from this it should communicate with other operation units and maintain a overview of the current cluster condition.

The system node itself is directly responsible for creating actor system with all required configuration options.

A configuration must be provided for the system node to start. The start list of modules is necessary together with module registration routine. The module registration is done on the source code level as it is not a subject of change in the runtime.

The system node is also responsible for creating the CI container and filling it with initial data.

The stop action is the only routine that the system node provides. These action signals to the cluster that the node is going down and all the services are shut down.

**Command Handler Support** The Command Handler Bus (CHB) is a prepared Module that can be registered as a dependency to a DI container. To CHB a command handler can be attached for a specific command type. Commands are DTOs that are created from deserialization.

The command DTO is forwarded to a specific command handler and it is processed there. The command handler returns success or failure. In the case of success, all the generated events are collected and passed to the event store.

There is also the support for repository type objects which allow to load an aggregate from event stream or to list all available aggregates.

**Event Support** Event support is mainly composed of event store which is able to serialize event DTO. These DTO come with a unique type to resolve correct DTO in the deserialization process.

**Projection Support** When the events are saved to the store, they are further send to processing in the projections. Each projection is registered in a central projection component.

Every transmitted event is forwarded to projection and it is completely up to the projection how it will react.

The framework offers a support for using SQL or NoSQL write support.

## 8.3 Testing Support

Every part of the framework is testable either as an actor (with trap-actor) or unit testable. To keep this promise a more functional style of objection design must be used. Every method should return a value. This value is significant at least for testing.

## 8.4 Self-Healing

Each module has a special supervisor actor that can return several case classes resulting in restart of the module.

```
class ModuleSupervisor extends ModuleRef with SLF4JLogging {

        def receive = {
                case StartNewModule(name, props) =>
                        log.debug("Registering module: " + name \
                                 + ", sender: " + sender.toString())
                        if (context.child(name).isDefined) {
                                log.error("Child already defined: " + name)
                        } else {
                                context.actorOf(props, name)
                        }

                case Forward(name, msg) =>
                        moduleRef(name) ! msg

                case StopSilentlyModule(name) =>
                        context.stop(moduleRef(name))
        }
}
```

The module should choose the action with care as the number of resources is always limited and can't for example add more processing units to the pool of workers indefinitely.

## 8.5 Integration

A very important parameter must be judged during the implementation. The framework must be able to cooperate with another existing framework. There are multiple major scenarios that need to be supported.

**Benefits for Existing Application** The framework provides a lot of benefits if added to the application. There are existing applications which need only a part of their functionality to be highly available or distributed.

The framework adoption should be as easy as just starting operation unit with some configuration from the existing application. A strategy for integration must be adopted, if the existing application is not distributed, it would make sense to start different type of

node with inside the application and completely different nodes on other virtual machines etc.

**Integrating Other Frameworks** The framework is designed to run a distributed network of modules that communicate with each other. This is the main purpose of the framework from a technical point of view and this is the only area at which the framework should be excellent.

There are many more areas that have to be covered like messaging, storage access etc. For each of these technologies the framework should make no presumption about the best suitable implementation. It's always up to the application to decide if the file storage service would be best served by GridFS or if a custom implementation using completely new file storage framework should be used.

The framework is limited only to the areas that it can do well. For every other area there is already an existing framework or solution that can be explored and used.

In a current technology setup there is currently no web tier solution chosen. Yet given a typical cloud application characteristics, it is very likely that some Model-View-Controller framework or other solution will be used in conjunction with the developed framework.

# Chapter 9

# Demo Application - Distributed Web Crawler

A framework is only as good as applications that can be built upon it. There is no better measurement of the framework overall quality than creating a demo application with similar requirements as the framework offers.

I have chosen to design and implement a distributed web archiving crawler. The crawler will „crawl“ over an input website, collecting all web pages and basic resources. These will be stored in an archive effectively creating a simple history for certain website.

The main idea is to create an online archive for web resources. The application will fulfill two main goals:

1. Incrementally collect and recollect content
2. Provide a browser of the web page history

The demo application is challenging, because there are many fields where the cloud environment can significantly improve the application infrastructure. The application needs to be massively scalable as it is clear that internet crawler can work in a massive parallelism. The application user interface can stay simplistic.

## 9.1 Overview

There are several main areas that need further analysis.

**Crawling over Web Pages** The process can be easily imagined as a pipeline of specialized tasks.

The main function of the application is to keep a queue of requested URL. Each URL is downloaded, parsed, processed and more referenced URLs are stored for download to the

queue. The downloader component can have a large pool or download workers running. The result of the downloader is a downloaded file for the queued URL.

The resources are stored to the database. Serialization occurs and the original document must be compressed if possible. The compression is necessary, because a large amount of data is expected, a large amount of documents will be of content type *text/html* or similar. The compression is very efficient for such plain text content types. Along with resources a parsed representation, it is stored also. For a web page this could be simply a page title and URL that appear on the page in the form of links. The parsing process is done by a special component and is highly dependent on the content-type of the stored document.

The inter-connection of the document is done by a URL that is supposed to be unique in a given time.

**Collecting Changes** The indexed URLs are scheduled for a check periodically. Same process as in indexing is applied and the domain may be crawled once again. A large amount of pages have dynamic content that is generated on the fly as the web page is rendered. These changes are not important for an archiving process. Each new copy is treated as a new entry in the database.

**Browsing Archive** A web server with a simple interface is exposed to end-users. A list of indexed pages is shown to the user. After selecting a domain the user is provided with a list of indexed pages on the selected domain.

Another feature of web interface is an overview of the queue of the scheduled URLs that are to be checked. An addition to this list is possible by a simple user input.

A last section of the web UI is the statistics overview. Some values are highly significant for each crawled domain for example:

1. Domain URL
2. Links references from the domain
3. Images references from the domain

## 9.2 Analysis

The analysis can be divided in two parts.

**Indexing Part** The indexing part of the application can be easily simplified to a pipeline of multiple processes. Each process can have multiple workers working in parallel.

This approach is well suited to be represented as modules:

1. DownloadHandler - manages the Download Queue and spawns downloaders from a Queue. The download can be successful in this case a Downloaded DTO is returned. In a case of failure (given by a HTTP error code) and error event is passed on.

2. Parse Module - is given a downloaded resources, a parser may be selected based on the content type

3. Storage Module - a module handling sending commands to the domain with a parsed document and compressed content

The Storage Module is responsible for the most logic. The domain is composed of only several entities:

1. DomainHost - representing a web domain that contains web pages and other resources

2. ParsedHtml - representing a content of URL with parsed links and images

A domain is notified about changes over the content. A new ParsedHtml can exist (a new URL is discovered) or a content may be deleted. The deletion is designated by a HTTP return code.

**Browsing Part** A browsing part has only one possible command - to append an URL for a download. No other commands can be issued from a web interface.

The information source for the browsing part is generated by several projections. A GraphDB approach is used for managing active domain links. A NoSQL storage has been chosen for ParsedHtml projections. This separation is for demonstrational purposes only, in a real-world application only one approach would be used. Another possibility for a storage facility is big-table engine such as Google BigTable or any other representative of this group.

To demonstrate working with files, a content of the web-page is stored inside a file storage facility.

## 9.3 Requirements

The application must be scaled easily across a large amount of nodes. There is only one instance of an indexing type node and one instance of a browsing part necessary to run an application. But in this configuration the application could be a traditional desktop application and would not benefit from the cloud environment.

There are two essential requirements for each application part.

**Indexing Nodes** Adding more nodes to the computational part of the cluster must be easy and the system must easily recognise these nodes. The nodes should be plugged instantly. Removing a computational node from a cluster should be equally simple.

The nodes should work in parallel, the computation should be load balanced. Multiple processes should work at the same time on each node as it is a logical approach to assume that each node running a virtual server would have its own network connection thus can utilize separated pool of resources.

**Browsing Nodes** Projections should run on database systems that provide high-availability and failover. The application should be simple and layered in order to minimalize the possibility of producing bugs. The application should merely display results of the database queries.

It is reasonable to assume that only two nodes are necessary for failover in case one virtual machine would go down. The web-server should be run as a cluster service forwarding every request to the browsing part of the application.

## 9.4 Implementation Overview

The references framework implementation was developed along this demo application.

**Used Components** The demo application is using a large amount of components that are well fitting the environment.

1. Bootstrap [29] - CSS framework
2. Spray [30] - Akka web framework
3. Jsoup [31] - parsing HTML
4. JGrapht [32] - module dependency graphing
5. Akka [33] - scala actor-based framework
6. Scalate [34] - templating
7. MongoDB [35] - NoSQL Storage
8. Neo4j [36] - Graph db
9. GridFS [37] - file storage
10. Lucene [38] - fulltext search engine

**Application Architecture**

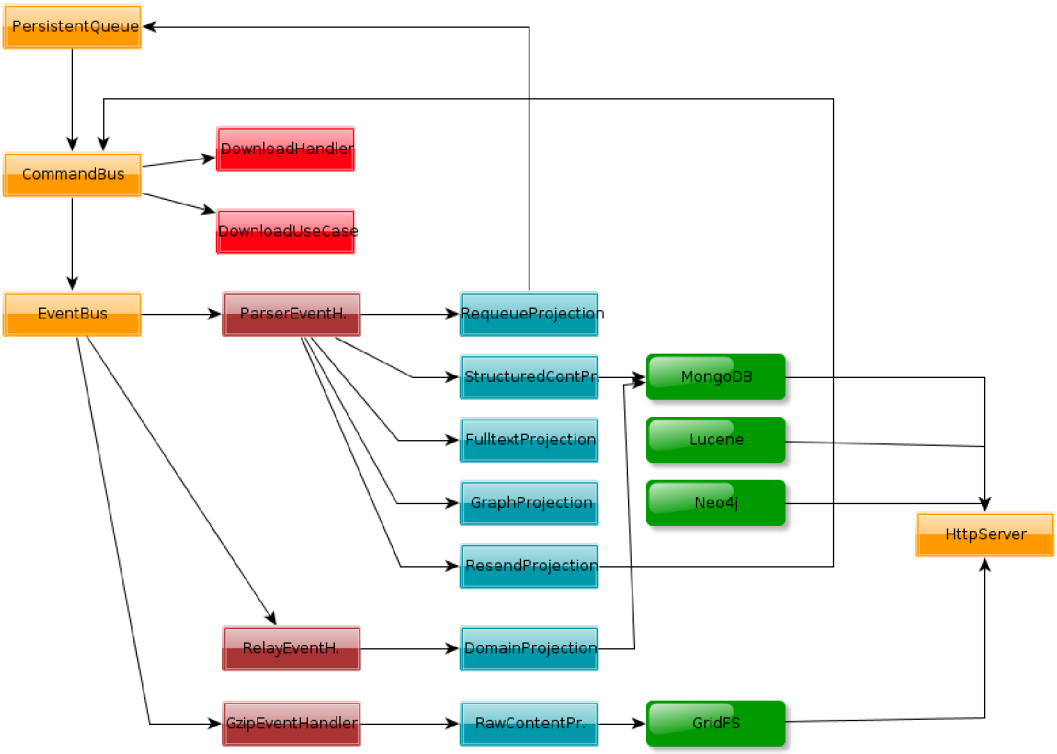



**Figure 9.1:** Mycelium Demo Application Architecture

## 9.5 Framework Benefits

The application is not trivial by far. Without a framework the task to implement such an application would be very challenging and would be difficult to accomplish. The quality of the product would be also at stake as the application would run in multiple threads and there would most certainly be synchronization problems.

The framework provides essential separation of logic for the application to several modules. Each module is an independent unit that can run multiple times thus providing a desired flexibility. Some services run only as dependencies, these are taken care of automatically by the dependency injection container.

The largest inovation and benefit of the framework are its scalability, clean separation of code and stability. Exactly these areas are highly valued in enterprise development and are crucial for a long-living mature server product.

The task to design such an application with a framework like this enables the developer to focus on the main areas of the application such as download process, parsing documents, storing them and maintaining projections. No other boiler plate has to be programmed, the communication is already done, the serialization too.

I believe that without the framework the application would never reach such a high quality standards. Simple usage of the framework has enabled a development with many essential tools that have already been taken care of.

## 9.6 Retrospective

The application is working according to the specification. Both of the tasks are fulfilled to full extend of the analysis.

The application speed can be scaled according to the number of nodes interconnected by the framework.

The designed framework helped to speed the development process significantly.

To better illustrate the application, screenshots have been collected in appendices of this thesis.

# Chapter 10

# Conclusion

This thesis has presented the foundation for building a high quality enterprise grade distributed framework. The design approaches are coherent and the main area of work is to connect them into several framework layers and principles that will allow developers to be more productive in building cloud applications.

I have analyzed the cloud environment. The requirements that has to be met in order to be successful in the cloud have been collected. A framework skeleton has been presented that has all the required abilities and in theory should stand to desired portion of various cloud application types.

The thesis has presented a reference implementation of the framework and has tested it on a medium sized enterprise-grade web crawler application. The web crawler can be seen as a typical example of extensively scalable cloud application. The application is functional and has been extensively tested in a multi-node environment.

The framework brings inovative ideas on how to compose the cloud application, analysis on the technology stack that can be used in the process. One of the benefits is also a reduced cost of cloud hosting as the framework is trying to provide maximal support for built-in technologies of cloud providers that are in common cheaper that general processor time usage.

## 10.1 Further Development Suggestions

In order to be successful, the framework must be released to a wider community and be developed further.

The original goals of the framework are accomplished, but there is a lot more work to be done in documenting the framework, enhancing stability, testing the environment in a long-term runs etc.

The main point of expansion should be an integration with more enterprise industry-standard technologies. The existing integration adapter is one of the most significant

factors that affect if the framework can be used or cannot be used in company products. If this ability of the framework is extended, it can be successful even in production environments.

# Bibliography


[1] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. [2013-03-15].

[2] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition).* Addison-Wesley Professional, anniversary edition, August 1995. [2013-03-15].

[3] Craig Walls and Ryan Breidenbach. *Spring in action.* Manning Publications Co., Greenwich, CT, USA, 2007. [2013-01-02].

[4] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley, 2010. [2013-01-02].

[5] Inc. Google. Google app engine. `s://developers.google.com/appengine/`, 2013. [2013-03-15].

[6] Microsoft. Windows azure. `http://www.windowsazure.com/en-us/`, 2013. [2013-03-15].

[7] Apache Software Foundation. Welcome to apache™ hadoop! `http://hadoop.apache.org/`, 2013. [2013-03-15].

[8] Jaroslav Čecho. Optimalizace platformy pro distribuované výpočty hadoop. Master's thesis, VUT Brno, 2011. [2013-03-15].

[9] Stříž Martin. Platforma pro vývoj ria aplikací. Master's thesis, VUT Brno, 2011. [2013-03-15].

[10] Bohačiak Ondrej. Programování s přístupem design by contract na platformě .net. Master's thesis, VUT Brno, 2009. [2013-03-15].

[11] George Reese. *Cloud Application Architectures - Building Applications and Infrastructure in the Cloud.* O'Reilly, 2009. [2013-01-02].

[12] Carson Gaspar. Deploying nagios in a large enterprise environment. In *LISA*. USENIX, 2007. [2013-01-02].

[13] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. [2013-01-02].

[14] Ben Kepes. *Understanding The Cloud Computing Stack - SaaS, PaaS, IaaS*. CloudU, 2011. [2013-01-02].

[15] Priya Viswanathan. Cloud computing – is it really all that beneficial? 2012. [2013-01-02].

[16] Benedikt Martens, Marc Walterbusch, and Frank Teuteberg. Costing of cloud computing services: A total cost of ownership approach. In *HICSS*, pages 1563–1572. IEEE Computer Society, 2012. [2013-01-02].

[17] Nic Laycock. How to build and lead successful online communities: What makes a community a community? *eLearn Magazine*, 2012(1):2, 2012. [2013-01-02].

[18] Sean Ludwig. *Cloud 101*. `http://venturebeat.com/2011/11/14/cloud-iaas-paas-saas/`. [2013-03-15].

[19] Inc. Google. Google compute engine. `s://cloud.google.com/products/compute-engine`, 2013. [2013-03-15].

[20] Inc. Amazon Web Services. Amazon elastic compute cloud (amazon ec2). `http://aws.amazon.com/ec2/`, 2013. [2013-03-15].

[21] Michael P. MacGrath. *Understanding PaaS - Unleash the Power of Cloud Computing*. O'Reilly, 2012. [2013-01-02].

[22] Inc. Amazon Web Services. Amazon web services. `http://aws.amazon.com/`, 2013. [2013-03-15].

[23] Shalini Ramanathan, Savita Goel, and Subramanian Alagumalai. Comparison of cloud database: Amazon simpleDB and google bigtable. 2011. [2013-03-15].

[24] Scott Urman. *Oracle PL/SQL programming*. Oracle Press (division of Osborne McGraw-Hill), pub-ORACLE:adr, 1996. [2013-03-15].

[25] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003. [2013-03-15].

[26] T. A. Cargill. A robust distributed solution to the dining philosophers problem. *Software—Practice and Experience*, 12(10):965–969, October 1982. [2013-03-15].

[27] Silvia Crafa. Behavioural types for actor systems. http://arxiv.org/abs/1206.1687, June 08 2012. [2013-03-15].

[28] Postgresql: The world's most advanced open source database. http://www.postgresql.org/, 2013. [2013-03-15].

[29] Bootstrap. http://twitter.github.io/bootstrap/. [2013-05-01].

[30] Elegant, high-performance http (and more) for your akka actors. http://spray.io/. [2013-05-01].

[31] jsoup java html parser, with best of dom, css, and jquery. http://jsoup.org/. [2013-05-01].

[32] Welcome to jgrapht - a free java graph library. http://jgrapht.org/. [2013-05-01].

[33] Akka. http://akka.io/. [2013-05-01].

[34] Scalate. http://scalate.fusesource.org/. [2013-05-01].

[35] Mongodb. http://www.mongodb.org/. [2013-05-01].

[36] Neo4j, the graph database - learn, develop, participate. http://www.neo4j.org/. [2013-05-01].

[37] Gridfs. http://docs.mongodb.org/manual/core/gridfs/. [2013-05-01].

[38] Apache lucene - apache lucene core. http://lucene.apache.org/core/. [2013-05-01].

# Appendices

# Appendix A

# Contents of the Enclosed DVD

The DVD enclosed inside the thesis contains the following files and directories:

- /thesis – thesis source code
- /mycelium – project source code

# Appendix B

# Running the Application

There is a detailed description of the building process inside the project directory in the README.md file.

Running the application requires at least:

1. RabbitMQ server (default user guest)
2. MongoDB (without authentication)
3. Running the application from the application root

```
./gradlew run
```

4. After booting the application can be reached on `http://localhost:8080/`.

# Appendix C

# Application Screenshots

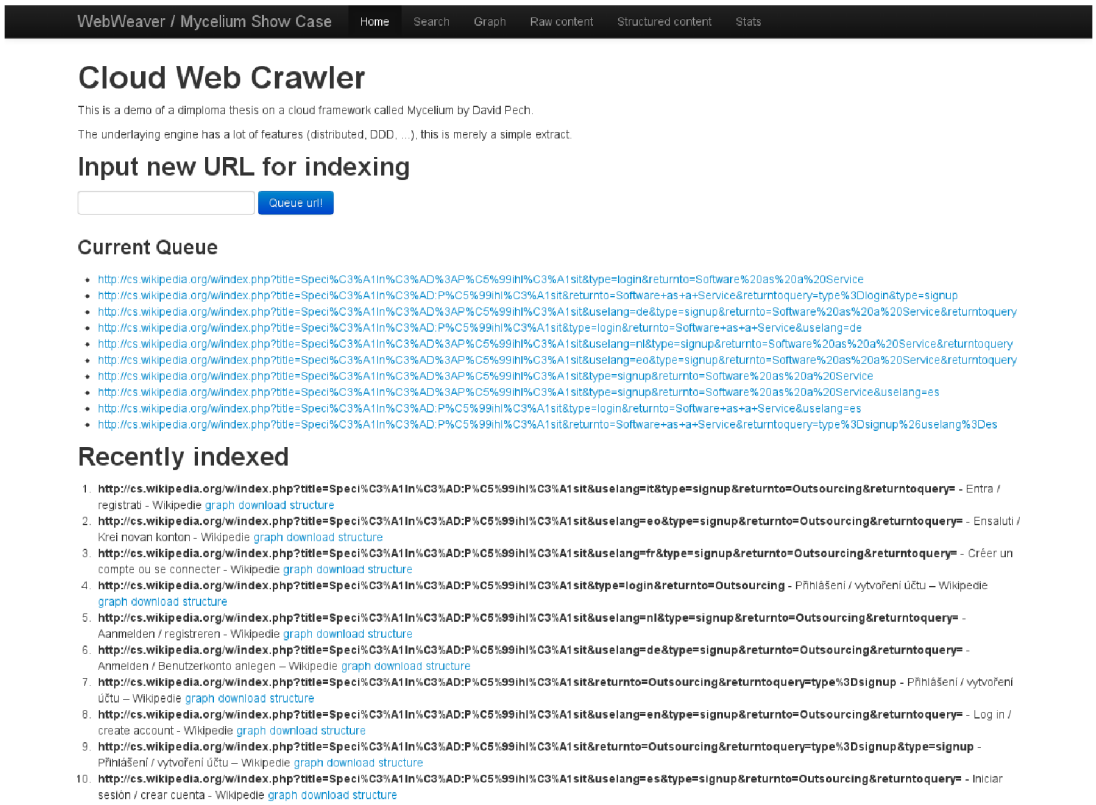



**Figure C.1:** Index Page with New Url Request

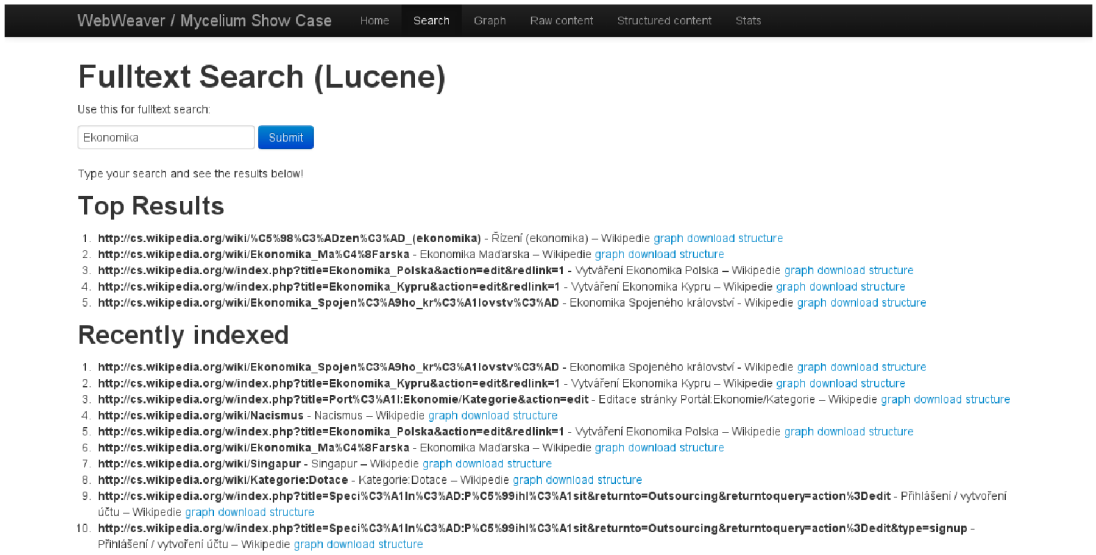



**Figure C.2:** Fulltext search using Lucene

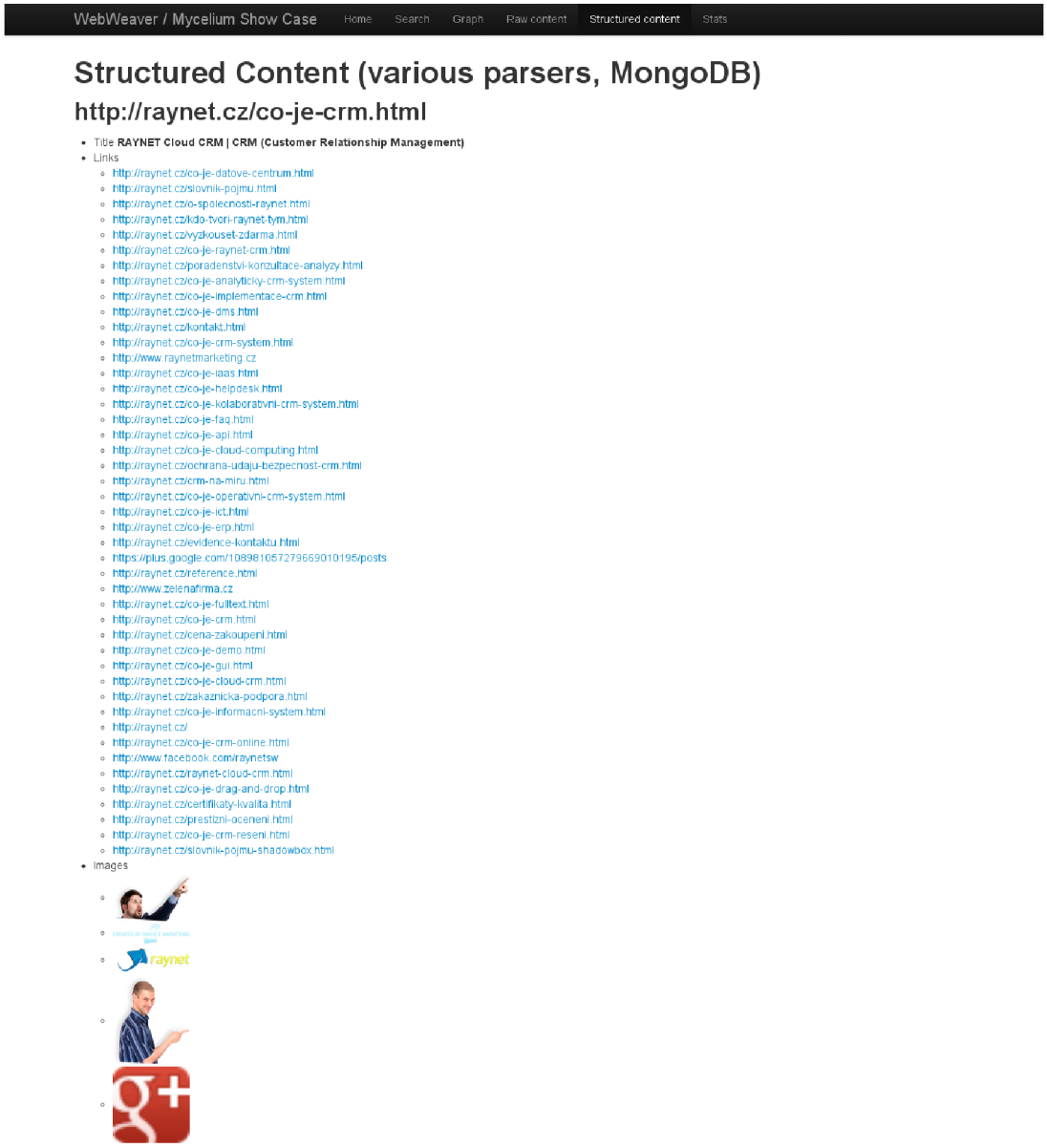



**Figure C.3:** Detail of a parsed page with links and images

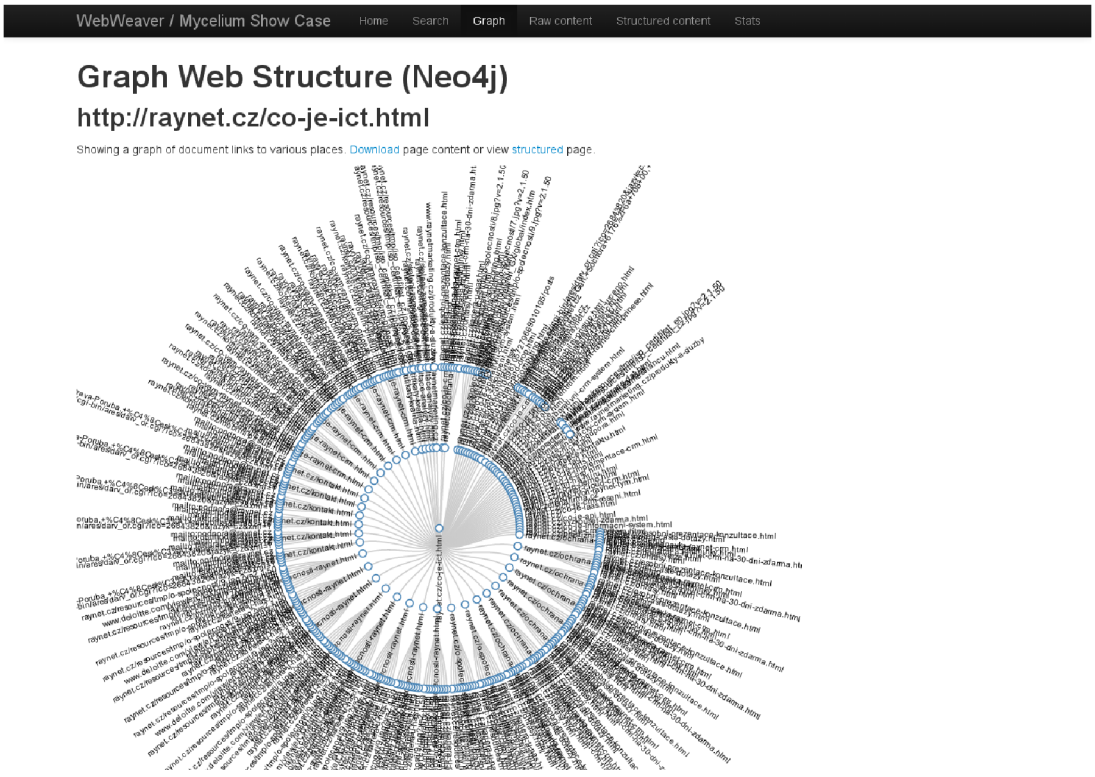



**Figure C.4:** Graph of links from a single page to others (2 levels)

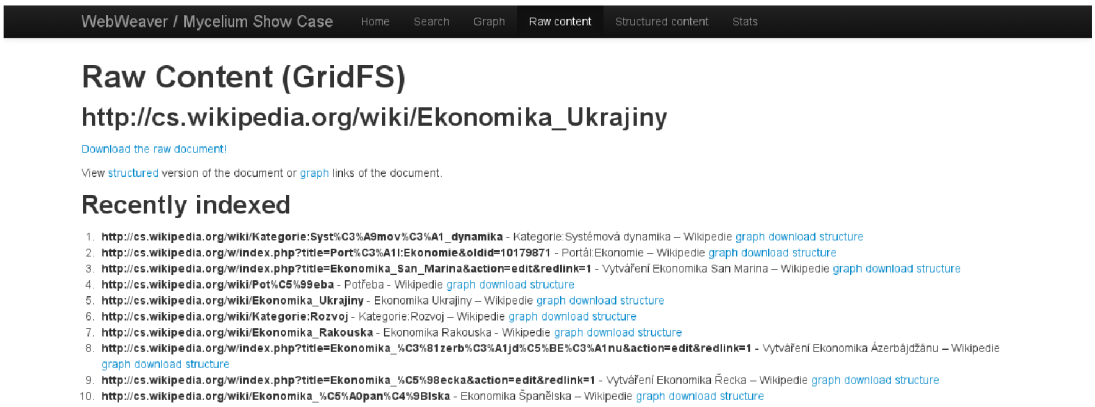



**Figure C.5:** Download of a raw (original) file

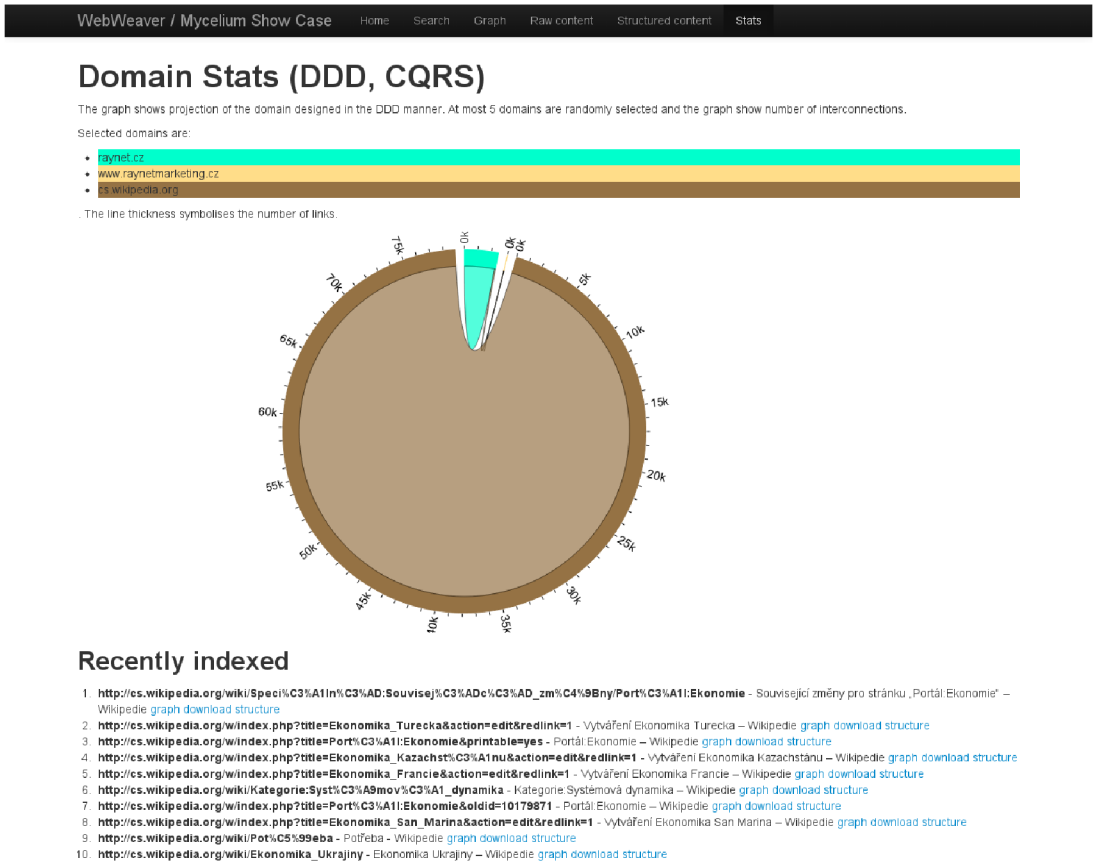



**Figure C.6:** DDD example - comparative number of indexed pages accross domains