University of Vienna
Faculty of Computer Science
Research Group Software Architecture

Semester Project

Distributed Calculation Cloud

2019S – 052500 VU – Distributed Systems Engineering

March 13, 2019

General Remarks

- This document covers the assignment of the **Semester Project**.
- You can achieve a total of **40 points** for this task.
- You must submit a mid-term **status update** (**SUPD**) before **06.05.2019 at 23:59**.
- The final project **deadline** (**DEAD**) is on **03.06.2019 at 23:59**.
  No deadline extensions are given.
- This is a **group assignment**. You and your work group members are allowed to work together in solving this task. Be aware that **all** group members are expected to contribute to the task and document their contribution.
- If you copy code or other elements from sources other than the lecture slides, please provide a reference to them in a comment above the corresponding entry.
- If you encounter problems, please post your question in the Moodle[1] discussion page, as it probably is also of interest to other colleagues. Alternatively, you can contact the tutors via dse.tutor@swa.univie.ac.at. As a last resort you can contact the course supervisor directly via dse@swa.univie.ac.at.

Submission Guidelines

All files required by this assignment have to be submitted to our GitLab[2] server into the proper project (repository) in the Submission and Feedback System[3]. **Be aware** to push your solutions into the correct submission **branch**, which is **2019s_dse_supd** for the status update and **2019s_dse_dead** for the final submission.

[1] https://moodle.univie.ac.at/course/view.php?id=105202
[2] https://lab.swa.univie.ac.at
[3] https://lab.swa.univie.ac.at/submission

As discussed in the Git[Lab Submission] Tutorial (GLST)[4] the submission branch is created for you. For any questions regarding the **GitLab**-based submission please refer to the GLST.

Plagiarism

We remind you that all of your submissions, implementations, designs etc. shall be created by you and your work group members. **Do not** use solutions from previous semesters, other groups, the Internet, or any other third party, as this would count as plagiarism and you would be directly graded with an "x" for the whole course. Code parts from Internet sites like Stack Overflow can be used but must be clearly marked as copied solution parts (put a comment both at the start and the end of each copied part).

Semester Project: Distributed Calculation Cloud

The goal of this project is to create a distributed calculation cloud application (akin to Google Genomics or the LHC Computation Grid at CERN) based on the concepts and technologies presented in the lecture. For this purpose, you will have to implement a solution that allows one to easily manage, integrate, and handle calculation services and related calculation tasks in a multithreaded and asynchronous fashion. This will require you to create service(s) which support different tasks (such as, to perform specific mathematical calculations). As it can be quite tedious to configure and monitor all such services (as real world projects would contain hundreds of them) you will need to add a service repository. Such a repository should enable other services to simply ask which Microservices, supporting a desired task, are currently available (i.e., similar to yellow pages — such concepts are also used at large companies, such as, Volkswagen). Further, a management services will enable to monitor, in a dashboard like fashion, the whole service landscape and provide means to initialize new calculation jobs. Finally, to prevent the uneven and excessive utilization of the provided computation power a billing service shall be established. The project will follow the Microservice architectural style[5] by applying Microservice patterns[6]. Accordingly, the system shall consist of five independent **Microservices (MS)**:

MS1 *Service Repository.* This service supports dynamically querying and registering Microservices and their endpoints (i.e., the services/endpoints created by your team) – such that each of the other Microservices solely needs to know the endpoints of **MS1** in advance – while all other MS endpoints can be queried dynamically. For this it must be possible to 1) register services/endpoints and service/functionality descriptions (i.e., how and which functionality, provided by a specific endpoint, can be accessed) and 2) query for a list of available services/endpoints which support a specific job. Basically, this Microservice should assume that not one but multiple Microservices can provide the same functionality (such as, specific calculations or billing) and the Service Repository should enable to programmatically identify Microservices/endpoints which support a desired functionality. Note that this Microservice must ensure that its data is

[4] https://lab.swa.univie.ac.at/submission/tutorial
[5] https://martinfowler.com/articles/microservices.html
[6] http://www.microservices.io

reasonably up to date (i.e., it should only contain services which are "likely" accessible at the moment). It should be extensively utilized by all other MS.

MS2 *Distributed Calculation Service.* This service provides means to offload and execute "long" running calculations, such as, data analysis tasks or machine learning jobs. In this course we will simulate such calculations based on basic arithmetic calculations. This service has to support at least the following calculations tasks: Addition, Multiplication, the calculation of Prime Numbers and Fibonacci Numbers. Each supported arithmetic task must be registered as an independent endpoint at **MS1** and implemented in a multithreaded fashion. Note that this Microservice must communicate in an asynchronous manner (keep this in mind when designing your network API). It should be possible to easily instantiate multiple independent calculation services simply by starting your **MS2** implementation multiple times.

MS3 *Billing Service.* This MS implements a billing service. As multiple instances of **MS2** could be running simultaneously it is necessary to motivate the users of this cloud infrastructure to spread calculation tasks fairly among all running calculation service instances based on the current load of each distributed calculation service instance. Otherwise one calculation service could be confronted with an overwhelming workload while others are completely idle – resulting in wasting energy and time. However, there could also be "valid" reasons why some users specifically want to solely use a single calculation service all the time. To resolve this challenge, we will use supply and demand to determine the price of the next calculation job execution for a given service instance. So, the price of offloading a single calculation task at a specific distributed calculation service instance increases/decreases based on the amount of calculation tasks which are simultaneously executed on that specific instance. This will require you to implement 1) user handling (to add, query, remove users, and their accounts) and 2) accounting logic (each user should have a simple virtual "bank account" which is charged for executing calculation tasks). The latter also requires that it must be possible to boost account balances based on **MS4**. Example: Assume that the user Tom was created before and his current account balance is 100€. Subsequently Tom sends a calculation task to a calculation service which is under heavy load. Hence, Tom is charged, e.g., 20€for executing his task (accordingly, his new balance becomes 80€). In comparison, if Tom would have sent the task to an underused service he would, e.g., only be charged 10€. It is sufficient if you store and manage accounts and balances based on **MS3** (e.g., the integration of a real world billing service is not necessary).

MS4 *Service Control Center.* This Microservice should represent a minimal control center for the outlined Distributed Calculation Cloud Application. For this it must enable to initiate the execution of calculation tasks while also enabling to interact with **MS3** (if you are a four-member team) to handle users (e.g., to add new ones) and their accounts (e.g., to query, display and increase an accounts' balance). Note that this is "only" a Microservice, hence, it can only be accessed via its designated network API (e.g., endpoints and network messages). Hence, a separate user interface must be provided in the form of a Command Line Interface (CLI) based client. The CLI client must be implemented as a standalone application which solely communicates with **MS4**'s network API (e.g., its endpoints). MS4, in return, contains the logic to communicate with the other Microservices accordingly. Examples are: to order the execution of a

calculation task, collect calculation task results, or to handle users and their account balances on **MS3**.

MS5 *Service Dashboard.* This service should provide a minimalistic and not necessarily good-looking web-based (i.e. HTML/CSS/JavaScript via HTTP) dashboard interface for the Distributed Calculation Services landscape. Through this **MS5** should provide a rough overview on the service landscape (e.g., service availability and calculation task performance indicators). Note that all the logic used by **MS5** should be located in a Microservice while the web-based dashboard solely provides a thin UI layer on top.

All team members are responsible for the design and implementation of all Microservices (out of **MS1-5**). Accordingly, we recommend that you apply pair programming, face-to-face discussions, multiple design iterations, and encourage all colleagues to bring in their skills and knowledge. This is especially beneficial as all Microservices need to collaborate and communicate. So that the envisioned communication protocol and API represents the needs and requirements of **all** Microservices, e.g., because some Microservices need to provide functionality and APIs which are not necessary for their own core functionality but solely to ease/support the work of another Microservice. If you don't have a fourth team colleague ignore and don't implement **MS3**. Note, that you, as a team, are responsible for the whole project and all project outcomes (e.g., the Microservices and their API). Accordingly, if there are team building or communication issues inform us, based on GitLab issues, in time. In detail, your responsibilities include the following:

Design and Collaboration. Think about the service's external interfaces: which functionality is provided to consumers of your services. Which API and protocol(s) should be used? Discuss and coordinate with your team colleagues. Each of you should prepare their own drafts/solutions/design ideas to form a basis for discussion, enable to spot missing parts, foster the exchange of knowledge and design ideas, and lay the foundation to create a suitable combination of all your ideas and concepts. Your project as a whole can only be successful if you learn to coordinate your efforts and collaborate efficiently within your team.

Implementation. Which technology stack will your service run upon? You can use Java for the implementation, but you are free to choosing any technology stack of your liking. In fact, Polyglotism (i.e. using different programming languages and technology stacks) is a major "selling point" of the Microservice architectural style.

Deployment. You will have to make sure that your service's functionality is made available to your team colleagues via a shared network (i.e. either a VPN or the Internet). In order to be accessible via a network, your service will have to 1) actually run somewhere (e.g. your notebook, your desktop at home, or even a hosted virtual machine, ...) and 2) listen on a port for incoming requests (e.g. a HTTP server listens on port 80).

Testing and Presentation. You have to make sure that each Microservice can be tested even when some/all other Microservices are not available. This is necessary as, typically, during development time not each service your implementation depends upon is available. So, be prepared for the project presentation (PRES) that your team might has to convince your supervisor that your working Microservices fulfil their requirements and work as expected, even if your team failed to implement some of the mandatory

Microservices. Think about suitable ways to present the functionality of your team's Microservices.

Error Handling. In dynamic MS landscapes errors can occur all the time and must be compensated (if possible) and communicated along the way to react accordingly. Hence, take error handling and error communication into account when designing your MS and the related network API. For example, how are you handling accounts which contain insufficient funds to execute a calculation task on a chosen calculation service?

Zero Configuration. MSs should build up their own network and interconnections dynamically, such that, it becomes, possible to simply start new MS instances which will automatically be picked up and integrated by all other already running MS instances. This is mandatory for real world Microservice landscapes which consists out of fifty or more different service types and hundreds of Microservice instances. Here, we simulate this concept in a simplified way based on **MS1**. Hence, you should use **MS1** to reduce the amount of hardcoded MS endpoints as much as possible.

Deployment

In the final project presentation (PRES) you will have to prove that your system is truly distributed. In other words, it should work even if each of your Microservices is executed/deployed on another physical machine (e.g. **MS1** is executed on team colleague A's notebook, **MS2** is executed on team colleague B's notebook, and so on. Make sure that your system can be deployed in such scenarios.

To overcome any routing/firewall/networking issues that might prevent your services from connecting with each other, we have prepared an OpenVPN server[7]. Upon connecting to this VPN server with your UNIVIE (u:account) credentials, you will be assigned a static private IP address. Our DSE VPN[8] Moodle page 1) explains how to connect with the VPN server and 2) contains a table that links student IDs to their corresponding IP addresses. You can use this list for looking up the IP addresses of your team colleagues.

Submission

Only material that has been submitted **in time** (pushed to your GitLab Work Group Project in the appropriate branch) and following the described *naming conventions* will be taken into consideration.

Status Update (SUPD)

2019s_dse_supd branch, deadline 06.05.2019 at 23:59

For SUPD you are required to deliver **class diagrams** (and skeleton like **implementations**) of your solutions (i.e., one or more class diagrams for each MS) along with a detailed definition of your network/microservice API (technology, endpoints, exchanged data, error handling, descriptions, and so on). We would recommend that you consider multiple design options

[7] https://openvpn.net/index.php/open-source/333-what-is-openvpn.html
[8] https://moodle.univie.ac.at/mod/page/view.php?id=2840482

for each major design decision and apply multiple design iterations. During each iteration you should execute and test your design on paper, e.g, by simulating different use cases to check if, e.g., your envisioned network API is really up to the given tasks. Based on our experience this will enable you to end up with a significantly better starting point for the final implementation phase (DEAD) than just using the first solution that crosses your mind. Ultimately, this will also reduce the amount of effort you will have to invest as each unlucky shortcut and overlooked error can create the need for substantial and effortful redesigns later on. Further, you should be able to defend each of your decisions during PRES.

It is expected that you provide, for each envisioned class, at least 2-3 sentences explaining its role and relations. The same should apply to each network message utilized by your network API.

The SUPD submission shall include the following actions taken:

- **each team member** has to connect at least once with our OpenVPN server.

The SUPD submission shall include the following artifacts in the root directory:

`implementation` A folder containing your **current** – (most likely) incomplete and/or broken – implementation (including javadoc, test cases, etc.). **Do not** bundle your code in archives like `.zip`, `.7z`, `.rar`, etc. The implementation for **SUPD** shall include **at minimum** the following:

- A **ms1** sub-folder of the current implementation of **MS1**
- A **ms2** sub-folder of the current implementation of **MS2**
- A **ms3** sub-folder of the current implementation of **MS3** (only required if your team consists out of four members)
- A **ms4** sub-folder of the current implementation of **MS4**
- A **ms5** sub-folder of the current implementation of **MS5**

`report_supd.pdf` A report on your **current** project status, with the following **compulsory** sections:

- A section **Microservice 1** of **MS1** with a short report about the current status of the Microservice with the following sub-sections:
  - A sub-section **Design decisions** of the Microservice. How does the service's external interface implementation look like? How can your colleagues use your service? Which protocol(s) and technologies are used? Describe your design and implementation approach, including one or more **UML class diagrams**, depicting the planned class layout.
  - A sub-section **Deployment** regarding your thoughts on the service's deployment. How are you deploying your Microservice? How can it be reached via the network, i.e. IP/hostname and port(s)?
  - A sub-section **Testing and presentation** containing the ideas for testing and presentation purposes. How are you going to test your service when other services, which you have to depend upon, are not available? How can you demonstrate the functionality of your service in such situations?

  - A sub-section **Status** describing the current state of your implementation. What has been achieved so far? What's missing? What works? What doesn't?

- A section **Microservice 2** of **MS2** with a short report about the current status of the Microservice with the same sub-sections as in Section **Microservice 1**.
- A section **Microservice 3** of **MS3** with a short report about the current status of the Microservice with the same sub-sections as in Section **Microservice 1**. Optional, only required if your team consists out of four members.
- A section **Microservice 4** of **MS4** with a short report about the current status of the Microservice with the same sub-sections as in Section **Microservice 1**.
- A section **Microservice 5** of **MS5** with a short report about the current status of the Microservice with the same sub-sections as in Section **Microservice 1**.
- A section **API specification**, following recommendations given in the article REST API Documentation Best Practices[9]. Strive to provide a detailed API documentation which should not only be applicable by your team members but also by third party developers which want to apply your documentation (e.g., to add additional Microservices or management clients) without needing a deep understanding of the internals, decisions, and interactions of your Microservices. For this you will need to describe exchanged messages, provide example data, describe potential use cases, related message sequences, and define how errors will be communicated/handled etc. Larger projects, such as, GitLab's API[10] or various RFCs[11] can be utilized as best practice examples.

The **Status Update** will be graded with up to **10 Points**, with up to **5 points** for the quality of your report and up to **5 points** for the extent and quality of your implementation.

In any case we strongly recommend that you have at least a skeleton implementation of your design (including the network interfaces/endpoints) ready by the **SUPD** deadline, so that afterwards you can focus on implementing the logic based on techniques learned in the course lectures.

Semester Project Deadline (DEAD)

2019s_dse_dead branch, deadline 03.06.2019 at 23:59

The DEAD submission shall include the following artifacts in the root directory:

`implementation` A folder containing your **final** implementation (including javadoc, test cases, etc.). **Do not** bundle your code in archives like `.zip`, `.7z`, `.rar`, etc. The implementation for **DEAD** shall include **all** of the following:

- A **ms1** sub-folder of the final implementation of **MS1**
- A **ms2** sub-folder of the final implementation of **MS2**
- A **ms3** sub-folder of the final implementation of **MS3**

[9] https://bocoup.com/blog/documenting-your-api
[10] Example API documentation GitLab: https://docs.gitlab.com/ee/api/projects.html
[11] Example API documentation RFC: https://tools.ietf.org/html/rfc4791

- A **ms4** sub-folder of the final implementation of **MS4**. Optional, only required if your team consists out of four members.
- A **ms5** sub-folder of the final implementation of **MS5**
- A working, tested (Unit Tests) and documented (e.g., JavaDoc for classes and methods) implementation of the outlined distributed system.

`report_dead.pdf` A single **PDF** file report on your **final** project status, with the following **compulsory** sections:

- A section **bird's-eye view** of your system. How does the system as a whole actually work? How do the pieces fit together? Describe how **MS1-5** are connected.
- A section **lessons learned**. What were your experiences in the project? What were the problems? Which problems could be solved, and which couldn't? Which decisions are your proud of? Which decisions do you regret? If you could start from scratch, what would you do differently?
- A section **team contribution**, which contains a **listing of contributions** for **each group member**.
- A section **Discussion**. How have you eventually deviated from your original plan laid out in SUPD? Which functionality could be implemented and which could not?
- A section **Interfaces & API** describing all your service's interfaces and your whole *final* API. For this you will need to describe endpoints, give example data and so on (see SUPD's description for details).
- A section **HowTo** documenting how the application is to be launched, initialized, and tested. A "quick start tutorial" that describes how one can use your services.

The final submission will be graded with up to **30 Points**, with up to **6 points** for the complete and correct report, up to **12 points** for the quality of your implementation, including the correct use of proper coding practices, and up to **12 points** for the completeness and correctness of your implementation in terms of executing the use cases outlined for the described Distributed Calculation Cloud.

Examination

- Each group has to present their solution at the end of the semester during the **Presentation Talk** slots (**PRES**). The assignment of groups to each slot will be announced in Moodle.
- The Presentation Talk is **essential** for the final grading and must be attended by all group members.
- Each group member should have knowledge of the entirety of the submitted solution, not simply the part he or she has worked on, and be able to explain the group's design process, decisions, and the rationale behind them.
- **No** elaborate presentation material (PowerPoint slides etc.) is required apart from the material you submitted as your solution.