**Task Forge**

**Development of a Distributed Client-Server Simulation Execution Framework**

**Candidat: Bogdan-Sergiu BOLOȘ**

**Coordonator: ș.l.dr.ing. Alexandru IOVANOVICI**

Sesiunea: Iunie 2024

# **REZUMAT**

În cadrul acestei lucrări va fi prezentat un sistem client-server, intitulat Task Forge, ce are în vedere execuția distribuită a unor simulări, conceput pentru a facilita gestionarea și execuția eficientă a acestora. Sistemul are două componente principale: partea de Client și partea de Server.

Partea de server este alcătuită dintr-o aplicație Flask care gestionează trimiterea sarcinilor, actualizarea de status, încărcarea și descărcarea fișierelor. Sarcinile de simulare sunt procesate folosind fire de lucru care execută sarcinile și utilizează funcții utilitare.

Partea de client oferă o interfață grafică utilizatorului care permite trimiterea sarcinilor de simulare și monitorizarea statusului acestora, la fel descărcarea rezultatelor.

Datele sarcinilor sunt gestionate într-o bază de date MongoDB, asigurând scalabilitatea în ceea ce privește gestionarea sarcinilor. Acest sistem permite execuția asincronă a sarcinilor, distribuită pe mai multe servere, sporind eficiența și fiabilitatea sarcinilor complexe de simulare.

**ABSTRACT**

This paper is going to be focused on the client-server system, titled Task Forge, which focuses on the distributed execution of simulation tasks. It is designed to facilitate efficient task management and their execution. The system is made of two main components: the Client side and the Server side.

The server side represents a Flask application which manages task submission, status updates, as well as file uploads and downloads. These tasks are processed using worker threads which execute the simulations using utility functions.

The client side provides a graphical user interface which allows the user to send simulation tasks as well as monitor them and download the results once the task is ready.

Task data is managed in a MongoDB database, ensuring scalability in task handling. This framework enables the asynchronous execution of the tasks, which is distributed across multiple servers, enhancing the efficiency and reliability of complex simulation tasks.

# **INTRODUCTION**

* 1. **Context**

Task Forge represents a robust client-server application developed to streamline the distributed execution of simulations tasks. The system is designed to handle the complexities associated with task management and execution in a distributed environment by leveraging the strengths of both client and server components.

The server side of the application is built using the *Flask* framework. *Flask* is a popular web framework for Python. The server component is responsible for managing the lifecycle of simulation tasks, from submission and status updates to handling file uploads and downloads. Tasks are processed through the worker threads, which utilize a suite of utility functions to execute simulations. The multi-threaded approach ensures that multiple tasks can be handled concurrently, thus improving the throughput and efficiency of the system.

The client side of the application provides a user-friendly graphical interface. This interface allows users to submit new simulation tasks, monitor their progress, and download the results once they are completed. The graphical user interface is built using *PyQt5*, offering a comprehensive set of tools for developing desktop applications.

Central to Task Forge is its use of *MongoDB* for task data management. *MongoDB* is a NoSQL database known for its scalability and flexibility. This makes *MongoDB* an ideal choice for managing the large volume of data associated with simulation tasks. By storing the task data in *MongoDB*, the application ensures that tasks can be managed and retrieved efficiently, supporting the system’s overall scalability.

The architecture supports asynchronous task execution, distributed across multiple servers. This design enhances the system's efficiency and reliability, ensuring that the complex simulation tasks are processed in a timely manner. The distribution of tasks across several servers makes Task Forge a powerful tool for users needing to execute and manage extensive simulation tasks.

* 1. **Motivation**

The development of this application was driven by the need to automate the process of executing simulation tasks, which previously required significant and frequent user intervention. For instance, users had to manually connect to a powerful computer via a remote desktop connection, initiate simulations, and subsequently transfer the results back to their own devices. This process was not only time-consuming but also prone to human error, leading to inefficiencies and potential inacurracies in simulation outcomes.

Task Forge addresses these challenges by automating and distributing the simulation process. The application eliminates the need for manual remote connections and file transfers, replacing them with a streamlined, user-friendly interface. Users can now submit simulation tasks through a graphical interface, monitor their progress, and download the results upon completion—all without the need for direct interaction with remote machines.

The motivation behind the project is also to provide a more efficient, reliable, and scalable solution for executing simulation tasks. By automating the process and distributing tasks across multiple servers, Task Forge significantly improves the workflow for users, allowing them to focus on analysis and interpretation of simulation results rather than managing the execution process. This transformation from a manual to an automated system represents a substantial leap in efficiency and reliability, ultimately enhancing productivity and accuracy in simulation-driven projects.

* 1. **Problem Statement**

Throughout this paper, I address the challenges and inefficiencies in the traditional manual process of executing simulation tasks. I will also delve into the architecture and functionalities of the client-server system by discussing its design, implementation and benefits. This paper aims to demonstrate how the system significantly enhances the efficiency, reliability, and scalability of simulation task management, ultimately transforming the workflow for users.

# **REQUIREMENTS AND SPECIFICATIONS**

**2.1 Theoretical Elements**

Throughout the development of this project, I identified several key functional and non-functional requirements to ensure the system meets the user needs effectively.

**2.1.1 Functional Requirements**

The client-side application needs to offer a graphical user interface that allows users to select the desired *tool* version, load simulation packages, submit simulation tasks, view logs of submitted tasks with their statuses, and download simulation results or receive notifications if results are unavailable.

On the server side, the system should handle task submissions, status updates, file uploads and downloads, track which server executes each task, and resume simulations after any workstation restarts. Additionally, it should allow users to redownload simulation results if needed.

**2.1.2 Non-functional Requirements**

The system must be scalable to handle many simulation tasks at once by distributing them efficiently across multiple servers. It should be reliable, ensuring tasks are completed even if there are system failures or restarts.

Performance is crucial and the system should be able to run multiple simulations in parallel depending on their size. The GUI must be easy to use, providing clear feedback and updates to users.

**2.1.3 System-level Requirements**

The project should include a mechanism to allocate simulation tasks to the least busy server, ensuring tasks are evenly distributed and completed on time. It should also maintain logs of all submitted tasks and track their execution status and locations.

**2.1.4 Hardware-level Requirements**

The system requires multiple servers to distribute the computational load, enhancing performance and reliability. Adequate storage capacity is necessary to manage simulation results, with a system to delete older results when storage is full and the ability to redownload results if needed.

**2.1.5 Software-level Requirements**

The server-side application should be built using Flask, which provides a robust framework for handling HTTP requests. MongoDB should be used to store and manage task data efficiently. The client-side application should be developed using PyQt5 to provide a rich, user-friendly interface. Integration with Artifactory is needed to manage and download different versions of *tool* required for simulations.

**Teorie despre tehnologii?**

**2.2 List of Requirements with Justifications**

For the system-level requirements, Task Forge needs a mechanism to allocate simulation tasks to the least busy server to ensure efficient distribution and timely completion. This mechanism should balance the load based on the size of the tasks and the current workload of each server. Maintaining logs of task submissions and their statuses, as well as tracking where each task is executed, provides transparency and accountability in task management.

At the hardware level, multiple servers are required to handle the distributed task execution enhancing performance and reliability. Adequate storage management is essential to prevent overflow and ensure the availability of simulation results. This includes deleting older results when necessary and allowing users to redownload results if needed.

On the software side, the server application should be developed using Flask, as it offers a solid foundation for managing HTTP requests and task operations. MongoDB integration ensures efficient management of task data, supporting scalability. The client application, built with PyQt5, provides an intuitive and responsive GUI for users. Lastly, integrating with Artifactory allows for easy management and downloading of necessary *tool* versions for simulations.

By meeting these requirements, Task Forge aims to provide an efficient, reliable, and user-friendly solution for managing and executing simulation tasks in a distributed environment, significantly improving the workflow for users.

# **DESIGN?**

Based on the functional and non-functional requirements I listed earlier, this section will focus on the project’s design.

As discussed in the previous section, the system is comprised of two main components: the client part and the server part, which are designed to make managing and executing tasks as efficient as possible in a distributed environment by working together. Using UML will help visualize and understand how these components work together to meet the goals of the project.

Unified Modeling Language (UML) is a visual way to represent and design a system. It uses diagrams to show the system's structure and how different parts interact.

Common types of UML diagrams include class diagrams, which show the system's classes and their relationships. Class diagrams provide a static view of the system by illustrating its classes, attributes, methods, and the relationships between the classes. Simply put, class diagrams are useful for visualizing the structure of a system and understanding how its components interact with each other.

**3.1 The Client**

*A diagram of a user flow

Description automatically generated*On the Client side, the main components handle user interactions and task submission.

**3.1.1 SimulationUtilities**

As its name implies, this class provides utility functions for determining the size of a simulation task based on its data. The main method here is *determine\_task\_size(data\_path)*.

**3.1.2 ArtifactoryUtilities**

This class interacts with Artifactory to retrieve and validate tool versions needed for simulations. It includes attributes like *ARTIFACTORY\_PATH* and methods such as *get\_tool\_versions()* and *is\_valid\_version(version)*.

**3.1.3 ConnectionHandler**

The “heart” of the Client system, this class manages connections to servers, handles task submission, and monitors task status. It has attributes like *servers* and *task\_server\_indexer*, and methods including *populate\_indexer()*, *get\_task\_indexer\_keys()*, *is\_server\_on(server\_url)*, *decide\_server()*, *get\_task\_queue()*, *add\_task(file\_path, tool\_version)*, *get\_task\_status(task\_id)*, and *download\_file(task\_id, save\_path)*. This class interacts with both SimulationUtilities and ArtifactoryUtilities.

**3.1.4 GUI**

The graphical user interface, this class is responsible for the interaction with the system, submitting tasks, as well as monitoring their status and eventually downloading them. It includes attributes such as *BASE\_URL*, *TASKS\_URL*, *STATUS\_URL*, *FILES\_URL*, *VERSIONS\_URL*, and *ARTIFACTORY\_PATH*. The methods in this class are *get\_crash\_count(data\_path)*, *determine\_task\_size(crash\_count)*, *get\_tool\_versions()*, *is\_busy()*, *get\_task\_queue()*, *display\_data\_path\_warning()*, *add\_task()*, *upload\_data()*, *send\_data()*, and *version\_changed(index)*. This class depends on ConnectionHandler for backend interactions.

**3.1.5 MainApp**

MainApp serves as the main entry point for the client application, integrating the GUI and connection handling and providing a CLI for quick interactions between the user and the system. It includes methods like *add\_task()*, *get\_task\_status()*, *and download\_file()*.

**3.2 The Server**

On the server side, the main components handle task management, processing, and file operations. The Flask-based server application receives task submissions, updates task statuses, and manages file uploads and downloads. Worker threads process the simulation tasks concurrently, ensuring efficient task execution.

A diagram of a software application

Description automatically generated with medium confidence

**3.2.1 Config**

Config handles configuration settings and database connections. Its attributes include *ROOT\_PATH*, *UP\_FOLDER*, *RECEIVED\_FILES\_DIR*, *mongo\_client*, *db1*, *task\_collection1*, *db2*, and *task\_collection2*.

**3.2.2 Server**

This class manages the *Flask* application, handling HTTP requests for task management and file operations. It includes attributes such as app and *task\_queue*, and methods like *is\_on()*, *add\_task()*, *get\_task\_queue()*, *get\_task\_status()*, and *download\_file()*.

**3.2.3 Task**

This class represents a simulation task, storing all relevant information about the task. It includes attributes like *start\_time*, *end\_time*, *user*, *data\_path*, *result\_path*, *version*, *size*, and *status*, and a method *set\_start\_time()*.

**3.2.4 TaskManager**

The execution of tasks, using worker threads to process tasks concurrently is all managed by this class. It has attributes like *task\_collection*, *pool*, and *active\_tasks*, and methods such as *start\_task()*, *task\_done(task\_id)*, and *manage\_tasks()*.

**3.2.5 Worker**

The *Worker* class represents threads that execute simulation tasks. It includes attributes like finished and *task\_collection*, and a method *run()*.

**3.2.6 SimulationRunner**

This class runs the simulations and manages the results directory. It has an attribute results\_directory and methods like *create\_results\_dir(param\_path)*, *search\_for\_rdb()*, *build\_exe\_path(version)*, *build\_param\_path(path)*, *open\_and\_extract\_zip(path, task\_id)*,

*run\_stools\_sim(exe\_path, parameter\_path)*, and *start(task)*.

**3.2.7 Utils**

Utility functions to support task execution and management are all found in this class. Its methods include *check\_for\_interrupted\_tasks(task\_collection)*, *create\_zip\_from\_result(results\_dir)*, *download\_tool(version)*, and *create\_tool\_dir(version)*.

The diagram and the enumeration of the classes show how each class is connected, emphasizing the interactions and dependencies within the system. By visualizing the structure, it becomes easier to understand how the client and server sides of this project work together to manage and execute simulation tasks efficiently.

TODO: interaction diagram