

**Project assignment from laboratory exercises in the subject:**

Technologie sieciowe

**Simple Service Mesh Management Protocol (SSMMP)**

**Major:** Computer Science

**Jakub Żurawicki**

**Jakub Bonda**

**Mateusz Kielak**

Siedlce 2023

# Topic of the analysis

The subject of the paper is to create the Simple Service Mesh Management Protocol (SSMMP) as a specification for deployment in a Cloud cluster, with the main goal of providing a platform supporting the required functionalities. The specification consists of message formats exchanged between protocol conversion parties (actors) and actions taken by message senders and receivers. The actors include: Manager, agents (residing on nodes forming the cluster), and instances of microservices operating on these nodes. The project must include elements such as registration and login handling, account management in the database, displaying the latest 10 chat messages in a table format, a chat enabling message sending and saving them in the database, and an HD file server allowing file upload and download. The project implementation will be carried out in Java, utilizing multithreading. The database will be created in MySQL.

Execution:

Jakub Żurawicki

Jakub Bonda

Mateusz Kielak

# Functional Architecture

### 

*Basic project schema*

# User Interface - Command Line Interface (CLI)

CLI - Command-Line Interface (CLI). It has the following functionalities (modes):

* **Registering a new user**
* **User login**
* **Adding posts (chat)**
* **Displaying the post board (chat): 10 latest messages**
* **File Transfer::**
* Sending files from a user-specified path to a special disk server for general filing.
* Downloading a file with a user-specified name.
* **HELP:** Basic information about the project and its usage.

# Manager

The **Manager** class is an integral part of the **ApiGateway** package. It serves as the central manager responsible for handling microservices within the application. This class follows the Singleton pattern to ensure that only one instance of **Manager** exists throughout the application's execution. Here's a breakdown of the key components and methods in the code:

**Fields:**

* **instance**: A private static field that holds the single instance of the **Manager** class.
* **microserviceAgent**: An instance of the **MicroserviceAgent** class, used for communication with microservices.

**Constructor:**

* **private Manager()**: This private constructor initializes the **microserviceAgent**, ensuring that only one instance of **Manager** can be created.

**Public Method:**

1. **getInstance()**: This public static method is used to obtain the single instance of the **Manager** class. It employs synchronization to ensure thread safety during instance creation.

**Private Methods:**

1. **handleCommunication(String request)**: This private method handles communication with microservices through the **MicroserviceAgent**. It sends a request and receives a response.
2. **registerMicroservice(String serviceName)**: This private method is responsible for registering a microservice. It creates a request string, sends it to the **MicroserviceAgent**, and parses the response to obtain the assigned port.
3. **parsePortFromResponse(String response)**: Another private method for parsing the port from the response string received after registration.
4. **parseRegisterFromResponse(String response)**: This method parses the registered port from the response string.
5. **findAvailablePort()**: This private method finds an available port for microservices. It starts from the default port number of 3000 and increments until an available port is found.
6. **isPortOccupied(int port)**: A private method that checks if a given port is occupied by attempting to create a new server socket. It returns **true** if the port is occupied.

In essence, the **Manager** class acts as a crucial component in the Microservice Manager, facilitating the registration, communication, and management of microservices within the application. It ensures that microservices are assigned unique ports and provides a centralized point for controlling their lifecycle.

# MICROSERVICE AGENT

The **MicroserviceAgent** class serves as the orchestrator for microservice management within the application's architecture. It handles the initiation, termination, and port retrieval for microservices. Here is a description of the key components and methods in the code:

**Fields:**

* **instance**: A private static field that holds the single instance of the **MicroserviceAgent** class.
* **microserviceThreads**: A map that stores microservice threads, indexed by microservice names.
* **microservicePorts**: A map that associates microservice names with their respective ports.

**Constructor:**

* **private MicroserviceAgent()**: This private constructor initializes the **microserviceThreads** map. It follows the Singleton pattern, ensuring only one instance of **MicroserviceAgent** exists.

**Public Method:**

1. **getInstance()**: This public static method is used to obtain the single instance of the **MicroserviceAgent** class. It employs synchronization to ensure thread safety during instance creation.
2. **startMicroservice(String serviceName)**: This static method starts a microservice with the given service name. It creates a new thread for the microservice and invokes its **main** method, allowing it to run independently.
3. **stopMicroservice(String serviceName)**: This method stops a microservice with the specified service name. It interrupts the microservice thread, releasing the associated port.
4. **communication(String request)**: This static method handles communication requests from the **Manager** class. It parses requests, extracts relevant information, and dispatches appropriate actions based on the request type.
5. **registerMicroservice(String serviceName, int port)**: This method registers a microservice with the given service name and port. If the microservice is not already registered, it adds it to the **microservicePorts** map and starts it.
6. **getPort(String serviceName)**: This method retrieves the port associated with a microservice based on its service name. If the microservice is not found, it returns -1.

The **MicroserviceAgent** class plays a crucial role in mediating interactions between microservices and the **Manager**. It enables the dynamic management of microservices by starting, stopping, and allocating ports, ultimately contributing to the overall microservices architecture's functionality.

# API Manager

The **APIManager** class in the **ApiGateway** package serves as the entry point for managing API-related functionalities in the application. It handles incoming client connections, communication, and interactions with microservices. Here is an overview of the key components and methods in the code:

**Main Method:**

* **main(String[] args)**: The entry point of the application, which starts the **APIManager** by invoking the **startAPIManager** method.

**Private Methods:**

1. **startAPIManager()**: This method initializes the APIManager, creates a ServerSocket, and enters a loop to accept incoming client connections. Each client connection is handled by a separate thread (**ClientHandler**).
2. **createServerSocket()**: Creates a ServerSocket on port 3000 and sets the reuse address property.
3. **acceptClientConnection(ServerSocket server)**: Accepts a client connection and returns the associated Socket.
4. **handleIOException(IOException e, String errorMessage)**: Handles IOExceptions by printing the stack trace and displaying an error message.

**Inner Class: ClientHandler**

* This inner class implements the **Runnable** interface and is responsible for handling communication with individual clients.

**Methods within ClientHandler:**

1. **run()**: Implements the **Runnable** interface's **run** method. It sets up communication streams with the client, handles client communication, and closes the client socket when done.
2. **handleClientCommunication(PrintWriter out, BufferedReader in)**: Handles communication with the client by processing incoming messages and responding accordingly. It identifies the message type and invokes the appropriate method.
3. **sendError(PrintWriter out)**: Sends an error response to the client in case of an invalid message type.
4. **sendMicroservicePortResponse(PrintWriter out, String serviceName)**: Sends a microservice port response to the client, requesting a port for a specific microservice. It communicates with the **Manager** to obtain the port.
5. **deleteMicroserviceAgent(PrintWriter out, String serviceName)**: Handles the deletion of a microservice agent and provides a log message.
6. **closeClientSocket()**: Closes the client socket.

The **APIManager** class serves as a central component for handling client requests and managing microservices' communication through the **Manager**. It provides a way to request microservice ports and handle agent deletions.

# API Gateway

**API Gateway** - CNApp entry points for users. Typically, an API Gateway has only one component. Its functionality involves forwarding user requests to the appropriate microservices. The API Gateway is stateless, operating without maintaining state (transferring byte streams between various components, both from microservices to clients and from clients to microservices). It maintains connections with clients using threads.

**Public Methods:**

**1. Method `run()`:** -The **`run()`** method is responsible for handling the client connection via the server socket. Upon accepting a connection, this method creates a **`BufferedReader`** object for reading input data from the client and a **`PrintWriter`** object for sending output data to the client. It then reads lines of requests from the client using the **`readLine()`** method and passes them to the **`forwardRequest()`** method for processing. Responses returned by **`forwardRequest()`** are sent back to the client using the **`PrintWriter`** object. In case of input-output errors (**IOException**), the method catches the exception and prints it to the standard error output.

**Private Methods:**

**1. Method `forwardRequest(String request, String serviceName, PrintWriter out)`:** The **`forwardRequest()`** method is responsible for forwarding requests from the client to the appropriate microservices based on the request type. Upon receiving a request from the client, this method registers and starts the corresponding microservice using the **`Manager`** class. It then retrieves the port on which this microservice is running and creates a client socket for communicating with the microservice. The request is forwarded to the microservice via the client socket, and the response is received and forwarded back to the client using the **`PrintWriter`** object. Finally, the microservice is stopped and unregistered using the **`Manager`**. In case of input-output errors (**IOException**) or thread interruption (**InterruptedException**), the method catches the exception and prints it to the standard error output.

Microservices in the application are also stateless, meaning they do not store state between requests and pass data (bytes) either to other microservices or to end users.

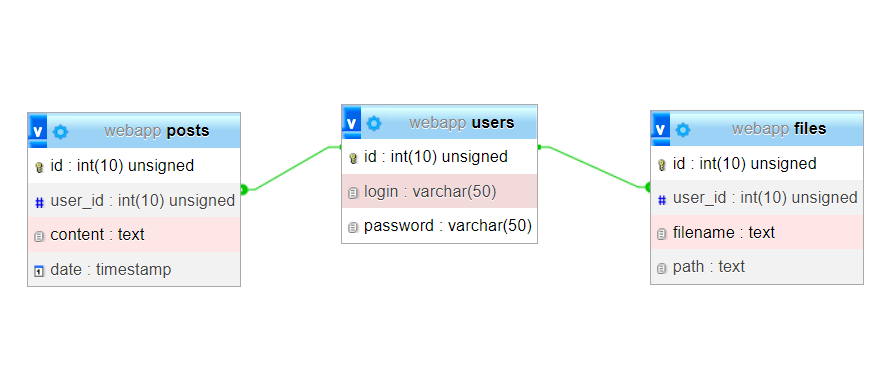
Types of microservices:

1. User login
2. User registration
3. Chat - available only for logged-in users, allows adding new entries to the board
4. Board - accessible without the need for logging in, displays the latest 10 posts
5. File transfer - available only for logged-in users, allows sending and downloading specific files.

**Database Project**

The database is written in MySQL and is named "ProjektTS". It is used to store information about users, files in the system, and archive user posts. It consists of three tables: "files", "users", and "chat". The detailed structure of this database is described below.

**Graphical Representation of the Database**

****

**Database code**

1. SET SQL\_MODE = "NO\_AUTO\_VALUE\_ON\_ZERO";
2. START TRANSACTION;
3. SET time\_zone = "+00:00";
4. /\*!40101 SET @OLD\_CHARACTER\_SET\_CLIENT=@@CHARACTER\_SET\_CLIENT \*/;
5. /\*!40101 SET @OLD\_CHARACTER\_SET\_RESULTS=@@CHARACTER\_SET\_RESULTS \*/;
6. /\*!40101 SET @OLD\_COLLATION\_CONNECTION=@@COLLATION\_CONNECTION \*/;
7. /\*!40101 SET NAMES utf8mb4 \*/;
8. --
9. -- Baza danych: `webapp`
10. --
11. -- --------------------------------------------------------
12. --
13. -- Struktura tabeli dla tabeli `files`
14. --
15. CREATE TABLE `files` (
16. `id` int(10) UNSIGNED NOT NULL,
17. `user\_id` int(10) UNSIGNED NOT NULL,
18. `filename` text NOT NULL,
19. `path` text NOT NULL
20. ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
21. --
22. -- Zrzut danych tabeli `files`
23. --
24. INSERT INTO `files` (`id`, `user\_id`, `filename`, `path`) VALUES
25. (2, 1, 'mysql-connector-j-8.2.0.jar', 'pliki\\login\\mysql-connector-j-8.2.0.jar'),
26. (4, 3, 'Czystarlinkimogastacsiealternatywadlagps.pptx', 'pliki\\loginn\\Czystarlinkimogastacsiealternatywadlagps.pptx'),
27. (5, 3, '380210059\_284015844511318\_3839805751918941074\_n.png', 'pliki\\loginn\\380210059\_284015844511318\_3839805751918941074\_n.png'),
28. (9, 2, 'l2z3.html', 'pliki\\login1\\l2z3.html'),
29. (10, 2, 'API.png', 'pliki\\login1\\API.png');
30. -- --------------------------------------------------------
31. --
32. -- Struktura tabeli dla tabeli `posts`
33. --
34. CREATE TABLE `posts` (
35. `id` int(10) UNSIGNED NOT NULL,
36. `user\_id` int(10) UNSIGNED NOT NULL,
37. `content` text NOT NULL,
38. `date` timestamp NOT NULL DEFAULT current\_timestamp()
39. ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
40. --
41. -- Zrzut danych tabeli `posts`
42. --
43. INSERT INTO `posts` (`id`, `user\_id`, `content`, `date`) VALUES
44. (1, 1, 'testowy post od logina', '2023-11-19 11:09:51'),
45. (2, 2, 'pozdro od login1', '2023-11-19 11:12:56'),
46. (3, 2, 'testtesttest', '2023-11-19 11:38:57'),
47. (4, 1, 'post testowy', '2023-11-19 17:15:49'),
48. (5, 1, 'post z rana', '2023-11-22 04:33:24'),
49. (6, 3, 'test o 6:10', '2023-11-22 05:10:31');
50. -- --------------------------------------------------------
51. --
52. -- Struktura tabeli dla tabeli `users`
53. --
54. CREATE TABLE `users` (
55. `id` int(10) UNSIGNED NOT NULL,
56. `login` varchar(50) CHARACTER SET utf8 COLLATE utf8\_polish\_ci NOT NULL,
57. `password` varchar(50) NOT NULL
58. ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
59. --
60. -- Zrzut danych tabeli `users`
61. --
62. INSERT INTO `users` (`id`, `login`, `password`) VALUES
63. (1, 'login', 'hasslo'),
64. (2, 'login1', 'hasslo'),
65. (3, 'loginn', 'hasslo');
66. --
67. -- Indeksy dla zrzutów tabel
68. --
69. --
70. -- Indeksy dla tabeli `files`
71. --
72. ALTER TABLE `files`
73. ADD PRIMARY KEY (`id`),
74. ADD KEY `user\_id` (`user\_id`);
75. --
76. -- Indeksy dla tabeli `posts`
77. --
78. ALTER TABLE `posts`
79. ADD PRIMARY KEY (`id`),
80. ADD KEY `user\_id` (`user\_id`),
81. ADD KEY `user\_id\_2` (`user\_id`);
82. --
83. -- Indeksy dla tabeli `users`
84. --
85. ALTER TABLE `users`
86. ADD PRIMARY KEY (`id`);
87. --
88. -- AUTO\_INCREMENT dla zrzuconych tabel
89. --
90. --
91. -- AUTO\_INCREMENT dla tabeli `files`
92. --
93. ALTER TABLE `files`
94. MODIFY `id` int(10) UNSIGNED NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=11;
95. --
96. -- AUTO\_INCREMENT dla tabeli `posts`
97. --
98. ALTER TABLE `posts`
99. MODIFY `id` int(10) UNSIGNED NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=7;
100. --
101. -- AUTO\_INCREMENT dla tabeli `users`
102. --
103. ALTER TABLE `users`
104. MODIFY `id` int(10) UNSIGNED NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=4;
105. --
106. -- Ograniczenia dla zrzutów tabel
107. --
108. --
109. -- Ograniczenia dla tabeli `files`
110. --
111. ALTER TABLE `files`
112. ADD CONSTRAINT `user\_idd` FOREIGN KEY (`user\_id`) REFERENCES `users` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;
113. --
114. -- Ograniczenia dla tabeli `posts`
115. --
116. ALTER TABLE `posts`
117. ADD CONSTRAINT `user\_iddd` FOREIGN KEY (`user\_id`) REFERENCES `users` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;
118. COMMIT;
119. /\*!40101 SET CHARACTER\_SET\_CLIENT=@OLD\_CHARACTER\_SET\_CLIENT \*/;
120. /\*!40101 SET CHARACTER\_SET\_RESULTS=@OLD\_CHARACTER\_SET\_RESULTS \*/;
121. /\*!40101 SET COLLATION\_CONNECTION=@OLD\_COLLATION\_CONNECTION \*/;

**General Operation Scheme:**

User Login:

1. The user enters their login credentials, i.e., username and password.

2. A request is sent to the login microservice through the API, and the data is appropriately interpreted.

3. The server authorizes this data by querying the "users" table in the database.

4. If the data is correct, the server grants access to resources only for logged-in users. The response contains the operation status, which, in turn, determines the user's login status stored in the "log" variable.

User Registration:

1. A new user enters their data, including username and password.

2. A request is sent to the registration microservice through the API, and the data is appropriately interpreted.

3. The server checks whether the login is not already in use by another user.

4. If the data is unique, the server adds the new user to the "users" table in the database.

5. A response is returned to the user with the appropriate status.

Chat:

1. The chat is available only for logged-in users. The "log" variable is checked.

2. If the previous point is true, the user can enter a message.

3. A request is sent to the login microservice through the API, and the data is appropriately interpreted.

4. Chat messages are stored in the "chat" table in the database, including information about the author, content, and time. The server sends a command to the database.

5. The status of whether the operation was successful is returned.

Board:

1. The board is available without the need for logging in.

2. A request is sent to the login microservice through the API, and the data is appropriately interpreted.

3. The server retrieves the latest 10 posts from the "chat" table that can be read.

4. In the response, the server provides the 10 latest messages to the user, and the data is interpreted and displayed in the console if the operation status is successful.

File Transfer:

1. Available only for logged-in users. The "log" variable is checked.

2. If the previous point is true, the user can perform the appropriate operation or send a request with the file name for download or the given file path for file upload, depending on the selected option.

3. A request is sent to the file transfer microservice through the API, and the data is appropriately interpreted.

4. The server connects to the database and retrieves values from the "files" table.

5. Then, from the server based on the saved URL address, the appropriate file is sent to the user by sending a response according to the protocols.

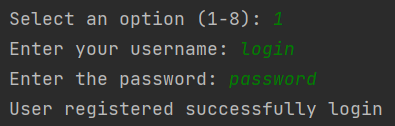
6. On the user's side, the file is decoded and the saveFile() function is called to save the file.

7. In the case of sending a file based on the given path, the file is encoded in Base64, and then a request is sent according to the protocols to the file transfer microservice where we interpret the request through the API and then send packets to the file server to save it in the folder. Data about the file and its URL address are stored in the database in the "files" table.

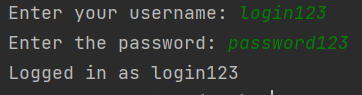
8. Finally, a message with the appropriate status is returned to the client.

**

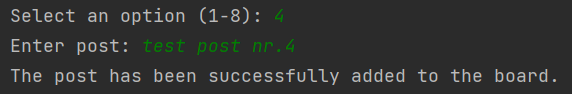
*Screenshot number 1 depicts the client-side application menu in English.*

**

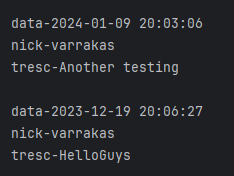
*Screenshot number 2 depicts the login from the client-side application menu.*

**

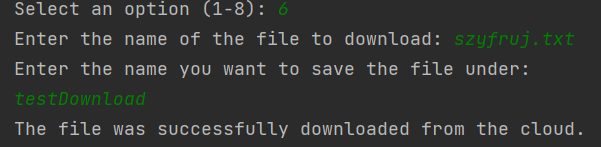
*Screenshot number 3 depicts the registration from the client-side application menu.*

**

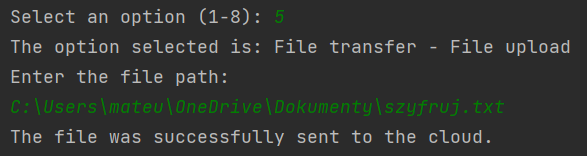
*Screenshot number 4 depicts adding a post from the client-side application menu.*

**

*Screenshot number 5 depicts the posts table from the client-side application menu.*

**

*Screenshot number 6 depicts downloading files from the client-side application menu.*

**

*Screenshot number 7 depicts sending files from the client-side application menu.*

Sources of information:

* Materials provided by Prof. Dr. Hab. Stanisław Michał Ambroszkiewicz
* Java. Basics. 12th Edition - Cay Horstmann

**THE END**

Computer Science UPH

3rd semester, year 23/24