Fall Semester 2025
Foundations of Distributed Systems
Part I: Communication
Lecturer: Prof. Dr. Christian Tschudin (christian.tschudin@unibas.ch, Office 06.002)
Assistant: Ali Ajorian (ali.ajorian@unibas.ch)
Exercise 1: Vector Clocks and gRPC (20 points, Uploaded: 02.10.2025, Deadline:12.10.2025)

Guidelines

This exercise consists of two tasks: the first involves an investigation of vector clocks, and the second focuses on working with gRPC, a remote procedure call framework. Both tasks require you to implement a code snippet, and you should adhere to the following guidelines throughout the process.

1. The corresponding exercise session for this assignment is scheduled for 07.10.2025, 10:15-12:00, in room 05.002, Spiegelgasse 5.
2. Implement your solution using Python 3.
3. Implement your solution in the provided code template attached to the exercise. **Do not change** the file structure.
4. Your implementation will be evaluated by running your scripts with Python 3. Submit all script files and resources you have used. If your code has dependencies, provide a **very brief but clear manual** in a *README.md* file explaining how to execute your code.
5. The use of Large Language Models (LLMs), including ChatGPT, Deepseek, Qwen, Grok or similar tools, is strictly prohibited for solving problems directly. While you may utilize these tools to enhance your understanding or gain knowledge, relying on them to solve problems will be considered plagiarism and will result in a penalty with a scaling factor between 0 and 1 on your grade.
6. This task must be completed in **groups of four** and only **one** should submit the implementation, acting as the team spokesperson. All submissions must be made **exclusively on ADAM**; submissions sent via email or any other medium will not be accepted.
7. The spokesperson must register the group by sending an email to ali.ajorian@unibas.ch with the subject **"FDS2025-teammembers-exercise1"**.
8. Sharing your solution with other teams is considered plagiarism for all members of both the sharing and receiving teams. In such cases, all participants involved will receive a **grade of 0** for the exercise.
9. Following grading, an interview will be scheduled and announced via email. All group members are required to attend together. Each member will be interviewed **individually** on questions from **both tasks**, assessing their theoretical understanding of the problem and solution, as well as their practical implementation work. Based on this interview, an individual contribution factor between 0 and 1 will be determined and applied to the team's grade to calculate the final individual grade.

Fall Semester 2025
Foundations of Distributed Systems
Part I: Communication
Lecturer: Prof. Dr. Christian Tschudin (christian.tschudin@unibas.ch, Office 06.002)
Assistant: Ali Ajorian (ali.ajorian@unibas.ch)
Exercise 1: Vector Clocks and gRPC (20 points, Uploaded: 02.10.2025, Deadline:12.10.2025)

Task 1: Vector Clocks (9 Points)

In this task, you will explore various methods of representing the causality of events within a distributed system. In Git context, each event (or commit) has a unique hash value that serves as an *event ID*. These event IDs are used as pointers within each commit object to reference other commit objects. This creates a Directed Acyclic Graph (DAG) data structure that illustrates the causal relationships between commits. On the other hand, vector clocks are vectors of N logical clocks that can determine the partial ordering of events in a distributed system with N processes. This means that these mathematical objects convey the causality between events without relying on edges as a DAG does. By doing this task, you will see how both DAGs and vector clocks convey and represent causality between events using different mechanisms.

Data Representation

For this task, we will represent Git DAGs using a JSON file formatted as follows:

```
{ "Branch_Name" : {"Commit_Name": list of parent commits}}
```

For example, the following JSON data represents a Git repository with three branches: B1, B2, and B3. In this structure, the commit [1] object 1111 in branch B1 has no parent, indicating it is the initial commit. Similarly, the commit object e13b in branch B3 has two parents, 9634 and f432. The accompanying DAG illustrates the complete causal relationships derived from this JSON data.

JSON Representation

```
{ "B1":
    {
        "1111":[],
        "12f3":["1111"],
        "f432": ["12f3","2101"]
    },
    "B2":
    {
        "2101":["1111"]
    },
    "B3":
    {
        "9634":["2101"],
        "e13b":["f432","9634"]
    }
}
```

DAG Representation

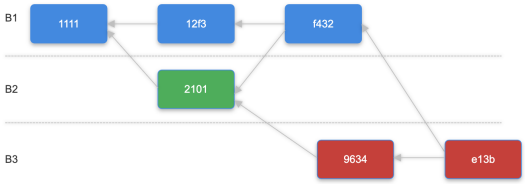

Figure 1: A Git repository represented by a DAG and a json file

Having the JSON file above, we can consider each branch as a process and each commit as an event. This allows us to calculate vector clocks for the commits, as illustrated in the following image.

[1] For simplicity, we have only written the last four digits of all hash values

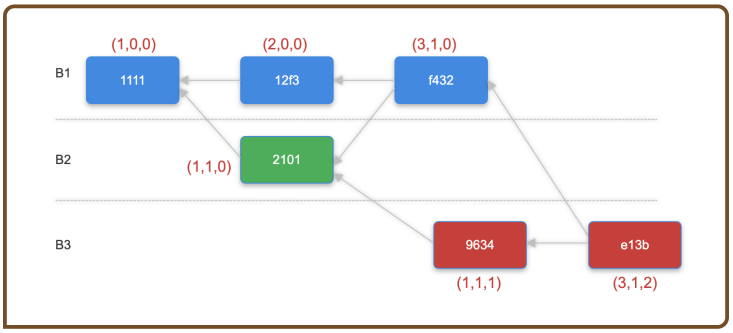

Figure 2: Vector clocks for the DAG represented above

Your Task

1. Given a JSON file representing a Git DAG as outlined above, write a Python script to calculate the vector clocks for each event, treating branches as processes and commits as events. You can store these values internally in a Python dictionary, as this will be helpful for subsequent tasks. However, the output of your function must be a JSON file formatted as follows (4 points):

   ```
   { "Commit_Name": list of clock values}
   ```

   The output of your function for the previous DAG should be formatted as follows:

   {"1111":[1,0,0], "12f3":[2,0,0], "f432": [3,1,0], "2101":[1,1,0], "9634":[1,1,1], "e13b":[3,1,2] }

2. Vector clocks establish a partial order relation among events. Implement a function named *causally_precedes(a,b)* that takes two vector clocks values, a and b, as input and determines whether the first vector clock precedes the second. Utilize this function and the output of section 1 to generate a graph where all nodes with causal relationships are interconnected. (2 point)

3. Some edges (causal relations) in the graph plotted in section 2 are redundant due to the transitive nature of causality. Develop a function that visualizes a graph with the minimum number of edges necessary to represent the partial order relation between nodes. To plot graphs you can use either Python libraries or graphviz utility. In the second case you should write a python script which generates a .dot file for each target graph. (3 point)

The JSON file in Figure 1 serves as a concrete example. Ensure that your implementation correctly handles different JSON files with similar structure.

Task2: Remote Procedure Call (11 Points)

In this task, you will explore gRPC, an open-source Remote Procedure Call (RPC) framework created by Google. It provides support for multiple programming languages, including C++, Java, Python, Go, Ruby, and Node.js, ensuring high performance and efficiency. To maintain language independence, gRPC utilizes Protocol Buffers (protobuf) as its interface definition language (IDL). This enables you to define the data structures and available methods for your service in a .proto file.

Theory of the Task

In today's distributed systems, it is common to delegate limited access to user data without exposing passwords to third-party services or applications. For instance, a user can register on a server, store their data in its database, and grant a third-party application access to certain parts of that data. To achieve this, various standards such as OAuth exist. However, for this task, we will create a simple protocol and implement it using gRPC as shown in Figure 3.

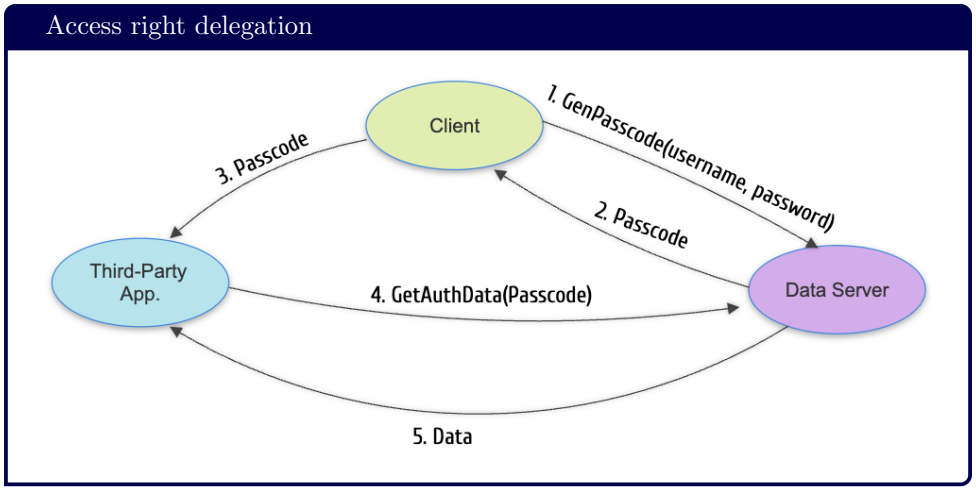

Figure 3: A user can grant access to their data using a passcode.

In our design, different users can register and store their text data in a *Data Server*. This server provides the following services:

1. A user can register on the server by providing a username and a password.
2. A user can store their data on the server by providing their username, password, and text data.
3. A user can retrieve their data from the server by providing their username and password.
4. A user can request the server to generate a one-time use passcode by providing their username and password.
5. A user or a third-party application can retrieve the user data by providing the one-time use passcode.

Having the features 4 and 5, a user can delegate their access to a third-party application. They must request the server to generate a passcode, which serves as a delegation tool, and then send this generated passcode to the third-party application. With the passcode in hand, the third-party application can retrieve the user data from the Data Server, effectively acting as the user's agent.

Target Scenario

As shown in Figure 4, our target scenario involves a third-party application that computes hash values for client data stored on a data server. The numbered steps in the figure indicate the sequence of operations. All components run locally: the data server on port 50051, the hash server on port 50052, and the client orchestrating the scenario. The data server services are already implemented in Node.js, while you must implement both the hash server services and the client using Python 3.

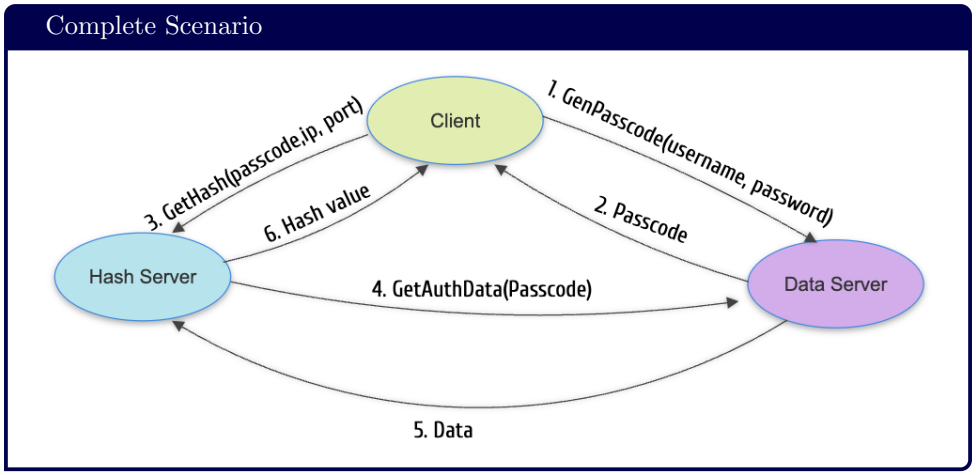

Figure 4: A third-party server can calculate Hash value of user's data

1. Data Server

The data server is pre-implemented in Node.js within the file *dataServer.js*, providing the gRPC data services described in the "Theory" section. The source code is located in the *dataServer* directory of the attached template.
To run the server:

1. Ensure Node.js is installed on your machine.
2. Install the required gRPC dependencies by running the following command in the *dataserver* directory:

   ```
   npm install @grpc/grpc-js @grpc/proto-loader
   ```

3. Execute the following command to start the data server:

   ```
   node dataServer.js
   ```

   Upon successful startup, you should see a confirmation message:

   `"Data server listening on port 50051".`

The data server implements the gRPC services defined in *dservice.proto* file:

```
syntax = "proto3";

package DATA;
```

```
message Result {bool success = 1;}
message Data {string msg = 1;}
message StoreReq{string username =1; string password = 2; string msg = 3;}
message Passcode {string code = 1;}
message UserPass{ string username = 1; string password = 2;}

service DB
{
    rpc RegisterUser(UserPass) returns (Result);
    rpc StoreData(StoreReq) returns (Result);
    rpc GenPasscode(UserPass) returns (Passcode);
    rpc GetData(UserPass) returns (Data);
    rpc GetAuthData(Passcode) returns (Data);
}
```

The services are as follows:

- **RegisterUser**: Takes a `UserPass` message (username/password pair) to register a new client on the data server.
- **StoreData**: Takes a `StoreReq` message (username, password, and data) allowing registered clients to store their data on the server.
- **GenPasscode**: Enables registered clients to generate a one-time passcode by providing their username and password. This passcode can be delegated to third-party applications (like the hash server) for temporary data access via `GetAuthData`. The passcode expires automatically after use.
- **GetData**: Allows clients to retrieve their stored data using their username and password.
- **GetAuthData**: Enables third-party applications to retrieve user data using a valid passcode obtained from the client.

2. Hash Server

The hash server provides a gRPC service for calculating hash values of text data. This service is used by clients who have previously registered with the data server, stored their data, and generated a passcode to delegate read access to the hash server. The service interface is defined as follows:

```
syntax = "proto3";
package Hash;

message Response {string hash = 1;}
message Request {string passcode =1;string ip=2; uint32 port = 3;}

service HS
{
    rpc GetHash(Request) returns (Response);
}
```

The **GetHash** procedure takes three parameters:

- A **passcode** generated by the data server
- The **IP address** of the data server (in our local scenario: `127.0.0.1`)
- The **port number** on which the data server is listening (port `50051`)

Using the provided passcode, the hash server retrieves the user's data from the data server, calculates its hash value, and returns the result to the user.

3. Client

The client performs the following sequence of operations:

1. Registers with the data server by remotely calling the `RegisterUser` procedure.
2. Stores data on the data server by remotely calling the `StoreData` procedure.
3. Generates a passcode by remotely calling the `GenPasscode` procedure on the data server.
4. Requests hash computation by remotely calling the `GetHash` procedure on the hash server, providing the obtained passcode.

Your Task

1. Run the data server on your machine.
2. Implement a Python server that provides the hash service defined in the specified gRPC prototype and run it (5 points).
3. Create a Python client that registers you on the *Data Server* using the username your_name.your_family_name [2] and your chosen password. Store your text data (less than 1KB) on the *Data Server* (3 points).
4. Request the *Data Server* to generate a passcode. After obtaining this passcode, connect to the *Hash Server* and retrieve the hash value of your data and output it (3 points).

gRPC on Python

Installation

To install gRPC on your python environment, use the following commands:

```
pip3 install grpcio grpcio-tools
pip3 install protobuf==5.27.2
```

To implement the client code and the hash server in Python, you need to compile the data server and hash server prototype files (*dservice.proto* and *hservice.proto*) into Python files. The following commands handle this compilation:

```
python -m grpc_tools.protoc -I. --python_out=.  --grpc_python_out=.  hservice.proto
python -m grpc_tools.protoc -I. --python_out=.  --grpc_python_out=.  dservice.proto
```

These commands generate the following Python files:

- *dservice_pb2py* and *hservice_pb2.py*: These contain the Python class definitions for our protocol buffer *messages* (like Request, Response, UserPass). They are used to create, serialize, and deserialize the data structures you send and receive.
- *dservice_pb2_grpc.py* and *hservice_pb2_grpc.py*: These contain the definitions for the gRPC clients (stubs) and servers (services). The *'*_grpc.py'* files provide the necessary classes to call remote procedures on the server and to implement the server-side logic that handles those calls.

[2] Any of the teammates or both can be registered

In short, the *_pb2.py* files handle your data, while the *_pb2_grpc.py* files handle the communication for their respective services.

Communication Basics

To interact with a gRPC server (such as the data server or hash server), first establish a connection by creating an insecure channel to the server's address and port, as shown in the following code snippet. This channel serves as the communication pipeline. Next, instantiate a stub (client) for the specific service you want to use; this stub provides local methods that correspond to the remote procedures defined in your *.proto* file.

```
import grpc
import hservice_pb2
import hservice_pb2_grpc
import dservice_pb2
import dservice_pb2_grpc

ip = "localhost"
port = 50051
channel = grpc.insecure_channel(f'{ip}:{port}')
d_stub = dservice_pb2_grpc.DBStub(channel)
```

To call a remote procedure, invoke the corresponding method on the stub, passing a properly constructed request message. For example, to call the *RegisterUser* procedure on the data server from the client, use the *d_stub* object as shown below:

```
usrpss = dservice_pb2.UserPass(username='John', password='Hello')
res = d_stub.RegisterUser(usrpss)
if not res.success: print('failure')
```

When a call is received by a server, the corresponding method is automatically executed, and whatever value it returns is sent back as the response to the client.

To start up a gRPC server so that clients can actually use its service, *grpc* module has a class *server* that can be used as follows:

```
    srv = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    route_guide_pb2_grpc.add_RouteGuideServicer_to_server(RouteGuideServicer(), srv)
    srv.add_insecure_port(f"[::]:{port}")
    srv.start()
    srv.wait_for_termination()
```

Lecturer: Prof. Dr. Christian Tschudin (christian.tschudin@unibas.ch, Office 06.002)
Assistant: Ali Ajorian (ali.ajorian@unibas.ch)
Exercise 1: Vector Clocks and gRPC (20 points, Uploaded: 02.10.2025, Deadline:12.10.2025)

References and Useful Links

- gRPG: A high performance, open source universal RPC framework
- matplotlib: a very powerful library for visualization with Python
- graphviz: a open source visualization software supporting visualization *language dot*
- nodejs: an open-source, cross-platform JavaScript runtime environment
- OAuth: an industry-standard protocol for authorization