Lecturer
Prof. Dr. Christian Tschudin (christian.tschudin@unibas.ch)

Teaching Assistants
Tom Rodewald (tom.rodewald@unibas.ch)
Ephraim Siegfried (e.siegfried@unibas.ch)

Assistant
Erick Lavoie (erick.lavoie@unibas.ch)

Uploaded on
Wed., 26. March 2025

Deadline
Sun., 10. April 2025 (23:55)
Upload to ADAM

Modalities

The exercises have to be done in **groups of 3 students**. Upload only the final version of the exercise to Adam and specify your group partners.

Please upload your short report, containing all answers to the asked questions as a .pdf and all code or script files to ADAM by the deadline at the latest.

For this exercise, you are required to write your solution using Bash scripts. Most of the tasks you will have to solve with git plumbing commands.

This exercise is best completed on a Linux machine or a Linux virtual machine.

Distributed Offline Chatroom with Git

The goal of this exercise is to familiarize you with offline-first communication in peer-to-peer systems by creating an offline-first chat application. Our application will use a single **public chatroom** model where all users write their messages on a public room. However, since it is a peer-to-peer application, there is no central server for message posting. Instead, we rely on data replication. Each user maintains a replica of all the messages they have received. These messages are stored in append-only linked lists, also known as append-only logs, with a separate log for each message sender. When a user wishes to post a message, they append it to the corresponding log in their local replica. Subsequently, when they are online, the user disseminates their entire replica, including all logs, to a subset of online users. Since all users engage in this behavior, over time, the logs of all users gradually synchronize. Through this process, all users ultimately establish a shared viewpoint of the chatroom, ensuring consistency across the system.
As Git provides local replicas with offline commit mechanism, we can take advantage of its infrastructure to build our application on top of that.

Causal Relationship of Messages: In order to capture the dependencies between messages, it is crucial to consider their causal order. For instance, when a user writes a text to the chatroom and another user replies to it, the second message is causally related to the first message. To maintain this causality across all users messages, each generated message should capture the chatroom's last state. This can be achieved by employing a graph data structure within the messages, where each message includes pointers to the set of the last sent messages from other users, known as frontiers. These pointers indicate that the posted message was created after the user had seen the frontiers set.

Assumptions: All users of the chatroom are trustworthy, meaning that:

- They don't create invalid messages.
- They don't post messages where they shouldn't.
- They don't tamper their replicas.

Design:

- **Repositories as local replicas**: Each replica is modeled as a Git repository.
- **Branches as append-only logs**: In each replica, a separate branch is created for each user from whom a message was received. Each branch is named after the corresponding user, allowing for easy identification and organization of the branches within the replica.
- **Commits as messages**: A message m would be a commit with the comment m and a tree object without any associated blob. So, when someone wants to send a message to the chatroom, they simply add a commit object to their local replica, placing it at the end of the branch that is named after them.
- **Causal Relationship as Parent Pointers**: To capture the causal order of messages, each commit would have all the latest messages from each log as its parent. This captures the most recent viewpoint of the user when they are posting a message.

- **Replication as Pull/Push**: The commands "`git pull`" and "`git push`" provide an easy way to disseminate an entire replica to a target user. By providing right arguments to these commands, all messages within a replica, organized in their respective logs, can be efficiently transferred and replicated in another replica, ensuring they are placed in the corresponding logs.

Documentation and Implementation: Please make sure to submit all of your implementations, including API functions and scripts, for the requested scenarios. Additionally, include a collection of screenshots that are accompanied by explanations indicating what they depict and to which specific task and subtask they are related.

Task 1 - Initialization (3 points)

(a) Implement the function `Initialize`, which takes a username (e.g., `<name>`) and creates a local Git repository. The initial branch should be named after `<name>`. The repository should contain a single tree object without blobs for the initial commit, and the commit message should be `<name> joined the chatroom`.

Note: In all of your implementations, the utilization of a blob object is unnecessary. Therefore, you should generate a tree object in the *Initialize* function that does not reference any blob. This tree object should be utilized in all subsequent commit objects that you create.

(b) After calling `Initialize` for the user `Alice`, list all objects in the repository by running `git cat-file --batch-all-objects --batch-check`. Then, check the repository's commit history using `git log` to confirm that only one commit exists.

(c) Initialize a repository for `Bob` in the same manner as for Alice. Establish one-way communication from Alice's repository to Bob's by executing `git remote add bob ../Bob` in Alice's repository. Use the `git push` command with the appropriate arguments to transfer Alice's commits to Bob. To verify that Bob has received Alice's commit, inspect Bob's repository using `git cat-file -p <hash>`.

(d) In Bob's repository, running `git log` (without arguments) may not display all commits from both Alice and Bob. Identify the issue, which may involve reference handling or branch pointers, and describe a solution to ensure that Bob can correctly see all commits.

Create a document that includes screenshots capturing the results of parts b), c), and d) along with the source code for your implementation of part a). Additionally, provide a detailed explanation of the issue encountered in part d), along with a corresponding solution.

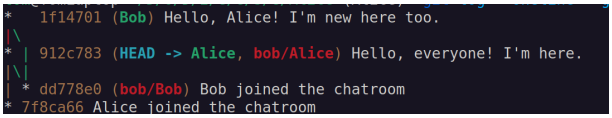
Task 2 - Posting Messages to the Chatroom (6 points)

(a) Implement the function `Connect`, which takes a path to a peer replica and adds the peer as a remote.

(b) Implement the function `Post`, which takes a string message as input and posts it in the chatroom under the name of the current repository as the name of sender user. To do this, you need to create a commit with the message content and the correct list of parents must encode the causal relationship between messages. The commit should then be appended at the end of the branch associated with the user who is posting the message.

(c) Implement the function `Push` that disseminates the messages to all users.

(d) Using your implementation for the functions *Initialize* , *Connect*, *Post* and *Push*, post a message *"Hello, everyone! I'm here."* from Alice to the chatroom. Then update Bob's repository and see Alice's message and reply to her, *"Hello, Alice! I'm new here too"*. Now update the Alice's replica and see the disseminated messages.
Using command "`git log --oneline --graph`" in Alice's repository, you should see an output similar to the following picture.



(a) *alice* branch

(b) *bob* branch

Figure 1: `git log` output for Alice's repository on different branches

Task 3 - User Interface (4 points)

(a) **Implement `Show`:** Write a function that outputs all chatroom messages in causal order with the format `"Author (time): message"`. You can rely on `git log` and parse or format its output accordingly.

(b) **Create a script:** Combine your `Initialize`, `Post`, `Push` and `Show` functions into a demonstration script where:

- `Alice` and `Bob` each initialize a repository.
- They exchange a few messages (using push/pull).
- `Show` is called to display the entire conversation from either side.

```
Alice (17:43 PM): Are you a student?
Alice (17:43 PM): I'm good, thanks for asking.
Bob (17:43 PM): How are you?
Alice (17:43 PM): Hey everyone
Bob (17:43 PM): Hey Alice!
Alice (17:43 PM): Alice joined the chatroom
Bob (17:43 PM): Bob joined the chatroom
```

(c) Modify the script you wrote in part (b) such that Alice changes her system time to an earlier point before posting the message "Hello everyone" and resets her system time afterward. When viewing the git log, does this change the order of the messages? Explain your answer.

(d) **Discussion:** Explain what happens if a new user `Carol` posts a message without first pulling from `Alice` (or other participants). Compare this scenario to what happens when `Carol` *does* pull the latest state before posting. Why is it important to merge or pull the most recent commits before appending new messages in distributed scenarios?

Useful Resources

- Git Internals - Plumbing and Porcelain
- Git Documentation
- Automatic time can be enabled or disabled using "`timedatectl set-ntp false`"
- Time can be adjusted using "`timedatectl set-time 23:55:00`"
- Time can also be adjusted using Timezones using "`timedatectl set-timezone Europe/Zurich`"