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., 09. April 2025

Deadline
Sun., 07. May 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.

Introduction

In this exercise, you will extend your chat application from the previous two exercises with additional functionalities based on state-based Conflict-Free Replicated Data Types (CRDTs). As presented in class, you will document your design by specifying:

(a) **partial order**: all possible states and their partial order;

(b) **merge**: the function used to combine two states, e.g. current state and received state;

(c) **monotonic operations**: all other operations that modify the state. You should argue that they all result in a newer state that is either equal or larger than the previous state.

Task 1 - UDP-Broadcast: Frontier Computation (4 Points)

Implement a standalone utility that tracks the number of messages each participant has sent in a distributed chat system. Each participant should maintain a *frontier*—a data structure that records the latest known message counts for all known peers. This frontier must be broadcast periodically (e.g., every 10 seconds). Upon receiving a frontier from another participant, the utility should merge the received state with its own to maintain a globally consistent view of all participants' message counts.
Suppose three participants—Alice, Bob, and Carol—are operating separate replicas. Their current states are:

- **Alice's view**: `Messages sent - Alice: 22, Bob: 35`
- **Bob's view**: `Messages sent - Alice: 22, Bob: 40`
- **Carol's view**: `Messages sent - Alice: 21, Bob: 38, Carol: 15`

After receiving updates from Bob and Carol, Alice's utility should display: `Messages sent - Alice: 22, Bob: 40, Carol: 15`. This indication should constantly be updated after receiving new updates.
Assumptions: In this task, a "message" is counted when a participant presses Enter in the input prompt. No actual chat messages are transmitted over the network. All participants have distinct usernames and broadcast only under their own names (no impersonation).

(a) Describe your state-based CRDT design according to its partial order, merge, and monotonic operations.

(b) Describe how the state of your CRDT is encoded in a single UDP packet with a maximum size of 576 bytes[1] and the associated limitations on the number of participants and the size of participants' user names.

(c) Create a file `task01.py` and implement your utility. Use the command line interface as defined in `utils.py`. Test the application with at least 2 machines to verify that all replicas are periodically consistent, i.e. they eventually show the same last recorded message count for all participants.

[1] See `https://stackoverflow.com/questions/1098897/what-is-the-largest-safe-udp-packet-size-on-the-internet`.

Task 2 - Frontier Computation with Persistent Storage (2 points)

In this task, you will enhance the functionality from Task 1 by persistently storing the frontier on disk and restoring it upon application startup. You will implement this persistence mechanism in a manner similar to how Git manages remote references using the `.git/refs/remotes` `directory` structure.
Each user must maintain a directory named `frontiers`. Inside this directory, there should be one subdirectory for each known participant p (including the user themself). Each of these subdirectories represents the local view of participant p's knowledge of the system. Within each participant's subdirectory, there should be one file per known peer p', containing the message count that p believes p' has sent.
This storage structure must be updated every time a message is received or the user presses Enter. Upon starting the application, the user should check for the existence of the `frontiers` directory. If it exists, the user should reconstruct their current frontier from the stored data — representing their best estimate of the global frontier at that time.

(a) Create a file `task02.py` and implement your utility. Implement the frontier storage and recovery mechanism as described. Use the command line interface as defined in `utils.py`. You can reuse code from Task 1. Test your implementation on at least two machines.

Task 3 - Git-based Chat (8 Points)

In this task, you will implement a Git-based chat application. You are required to extend the application from Exercise Sheet 2 by replacing the `push` and `connect` commands with your own implementation that leverages frontiers. The storage of messages remains the same as defined in Exercise Sheet 2.
When a frontier is received from another replica, your application should determine which messages the peer is missing by comparing it to the local frontier. The missing messages must then be propagated in such a way that the receiving peer can recreate the commit, i.e. the commits must have identical commit hashes. This mechanism should minimize redundant network traffic by only sending the necessary messages according to the received local frontiers.
Upon receiving new commit messages, replicas should commit these messages and update their Git references (i.e., `refs/heads/<PARTICIPANT>` for each corresponding author). Note that, unlike the previous tasks, the local frontier here should be updated **only** after a commit has been completed. It should not be merged with received frontiers; otherwise, other replicas may not send missing messages. For instance, the local message count for a user is incremented only upon receipt of a commit with that user as the author.

(a) Describe how a single message, stored locally as a Git commit, is encoded as a UDP packet and the associated limitations on the number of participants, the size of participants' user names, and the size of the content of a message;

(b) Create a file `task03.py` and implement your utility. Use the command line interface as defined in `utils.py`. You can reuse code from Task 1 and Task 2. Test it with at least 3 machines to verify that all replicas are eventually consistent, i.e. they replicate all the same messages and compute the same local frontiers, even if one of the replicas is temporarily offline.

(c) Imagine a practical scenario with several users in the chatroom, where one user has been offline for some time and comes back online after a while. What could be potential issues with your implementation and can you provide possible solutions for those issues?