# Task 3.1

(a) To answer the question on how our state-based CRDT design will be / is implemented, we discuss its partial order, merge and monotonic operations: For the partial-order we will regard two participants Alice and Bob. In our case by pressing "Enter" we create a new message that will be counted. If for example Alice and Bob create concurrent messages that gets updated, we may not have a defined ordering of the messages. Alice could have already created messages before Bob. Such a "Happens-before" process is a partial order. The merge operation will synchronize the chat state between multiple instances. The purpos of it will be to merge the local state with the incoming state recieved via the broadcast from another peer. The merge function should loop though each user and count in the incoming state. If the User does not exist in the local state or if the recieved count is greater than the current local count it updates the local state. When an update occures it should notify the user the new merged state. The monotonic operation will be also handled during the synchronization. The merge function ensures that the message count per user only increases or stays the same. A user's count will get updated only if the incoming count is greater than the current one. Otherwise, if the incoming count is lower or equal, it's ignored. This should guarantee that once a higher count is seen, it is never overwritten by a smaller value.

(b) The state is encoded as a JSON string and sent via UDP.
**Example**:

$$\{"alice" : 22, "bob" : 35, "carol" : 15\}$$

To ensure safe transmission over the Internet, the UDP payload is limited to 576 bytes. The JSON string must therefore not exceed 576 bytes.

**Number of participants**: Assuming each entry is approximately 20 bytes on average, the system can safely support around:

$$576\ bytes\ /\ 20\ bytes\ \approx\ 28\ participants$$

To be safe, limit to around 20–25 participants.

# Task 3.3

(a) A single chat message is stored locally as a Git commit and then transmitted as a UDP packet. Typically, a UDP packet contains the most important data for reconstructing a commit:

- **Commit hash (e.g. SHA-1)**: 40 characters
- **Author (username)**: Variable, but often limited to about 32 characters
- **Timestamp**: UNIX timestamp (e.g. 10 characters)
- **Message text**: Depends on the maximum packet size (e.g. 512-1024 characters)
- **Reference to previous commit hashes**: to guarantee the sequence (depending on implementation)

**Limitations**:

- **Maximum number of participants**: Since each participant has its own Git branch (refs/heads/<PARTICIPANT>), a large number of participants could lead to scaling problems.
- **Username size**: Limited by the UDP packet size; very long names can affect the structure of the packets.
- **Message content size**: UDP packets are often limited to a size of 576 to 1500 bytes. Long messages may need to be split or compressed.
- **No guaranteed delivery**: UDP is connectionless - packets may be lost or arrive in the wrong order. Therefore, commit references or checksums must ensure consistency.

(b) **Possible problems**:

- **Missing messages**: Since messages are synchronized via Frontiers, the user is missing all commit messages that were sent during their offline time.
- **Inconsistent history**: Other participants could already be based on a new commit history, which could lead to conflicts.
- **Network overload**: If the user catches up on many missing messages, a large data transfer could be necessary, which increases the network load.

**Possible solutions**:

- **Targeted synchronization**: when rejoining, the user should synchronize their frontier and only retrieve the missing messages instead of reloading all commit data.
- **Version control through Git**: Since Git is used, the user can only catch up on the relevant changes via `git fetch` or `git pull`.
- **Prioritized synchronization**: If many messages are missing, they could be prioritized based on timestamps (e.g. load newer messages first, then catch up on older ones).