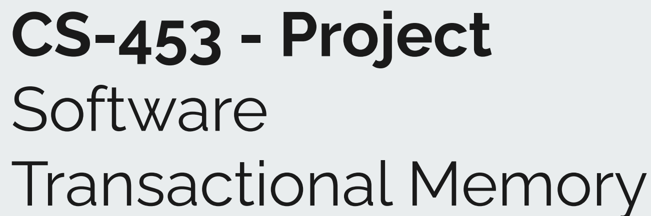


Distributed Computing Laboratory

September 16, 2025

In short:

- 30% of the grade of CC
- A single deadline
- Individual (YOUR code, no LLMs), but
- You can work in small groups
- Automated grader
- Reference document on Moodle
- Project sessions are Q&A

How does this project complement the lectures?

Lectures (mostly) focus on ***wait-free*** algorithms

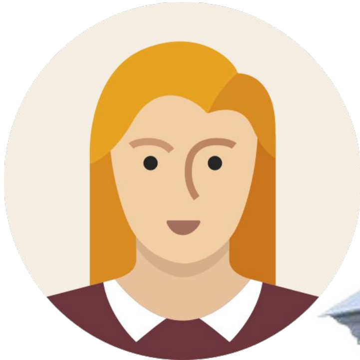
- **Never block**ed by other (slow/unscheduled/dead) processes, but
- **Tricky** to design/code
- Can be **expensive**
- **Difficult** for a **full application**

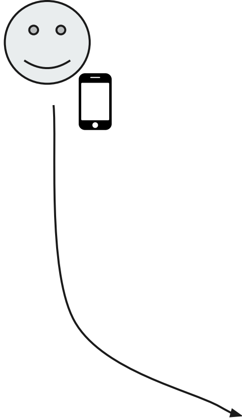
This project focuses on an alternative, ***transactional***, concurrency-control model, which:

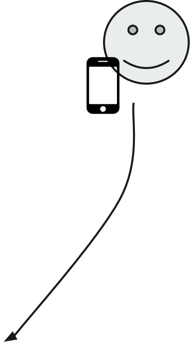
- **Sacrifices wait-freedom** (ok for most applications), but
- Is **easy to use** (forget about concurrency + never deadlocks)

Your task: implement a transactional framework to make concurrent computing easy!

Let me introduce you to Alice...



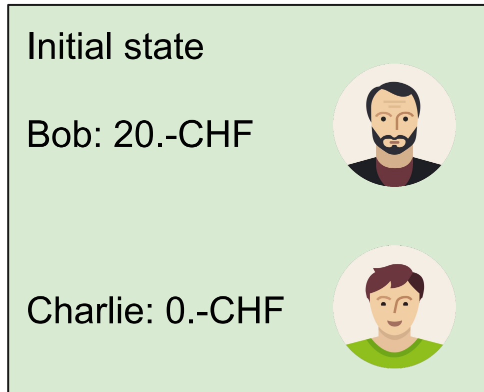


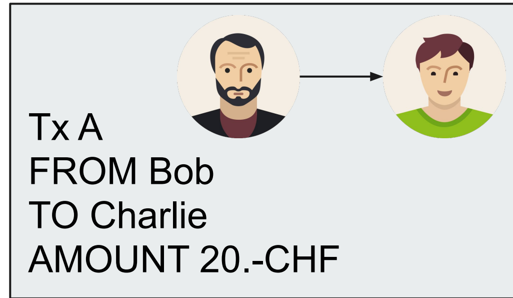


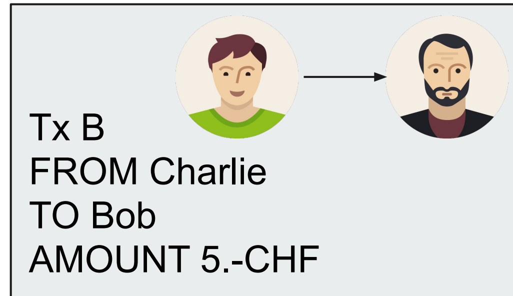
How Alice's Bank works

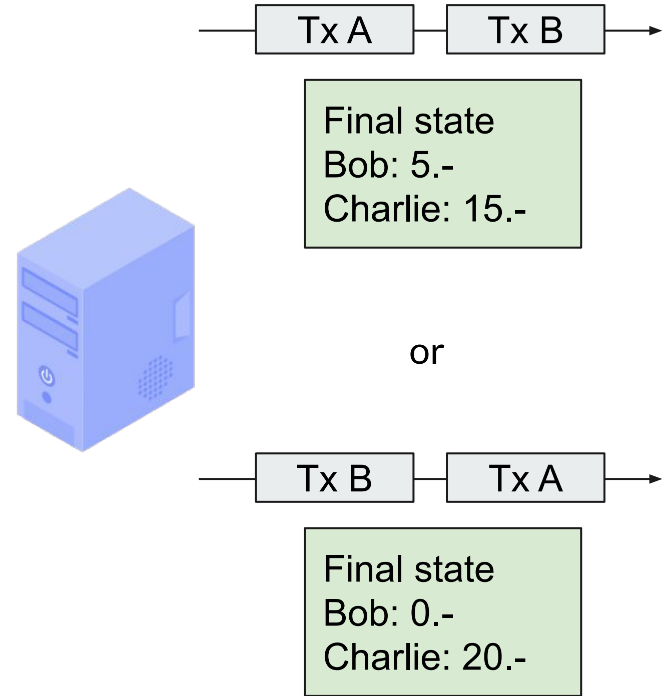
```
integer[] accounts;

fn transfer(src, dst, amount)  {
    if (accounts[src] < amount) // Not enough funds
        return; // => no transfer
    accounts[dst] = accounts[dst] + amount;
    accounts[src] = accounts[src] - amount;
}
```

Executions

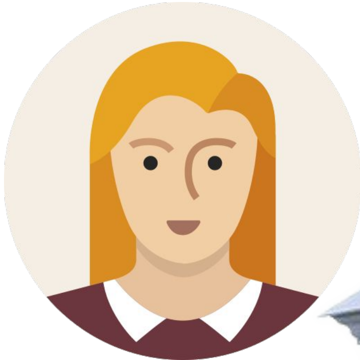


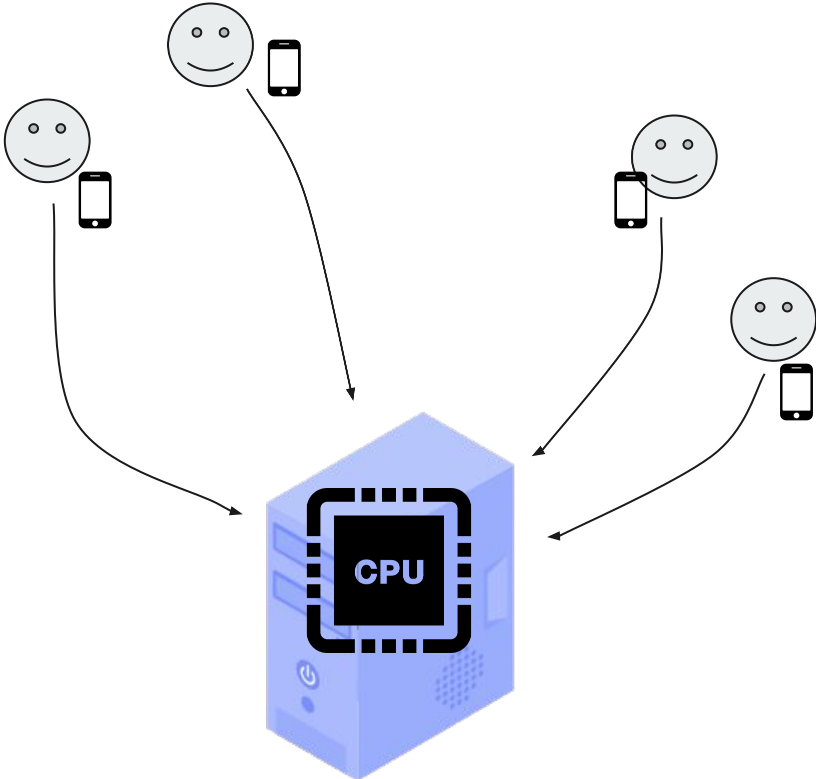




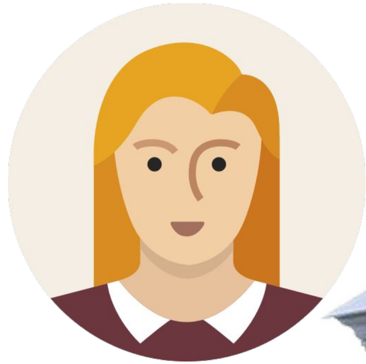


Scaling





Free lunch is over



-13.-CHF

-2.-CHF

OOOPS!

Closer look

```
integer[] accounts;

fn transfer(src, dst, amount)  {
     let v0 = accounts[src]; // 0th read
     if (v0 < amount)
          return;
     let v1 = accounts[dst]; // 1st read
     accounts[dst] = v1 + amount; // 1st write
     let v2 = accounts[src]; // 2nd read
     accounts[src] = v2 - amount; // 2nd write
}
```

Concurrent transfers

Initial state

Bob: 20.-CHF
Charlie: 0.-CHF

Final state

Bob: 25.-CHF
Charlie: 15.-CHF

Tx A
FROM Bob
TO Charlie
AMOUNT 20.-CHF

Read B → 20
Read C → 0
Write C ← 20

Read B → 20
Write B ← 0

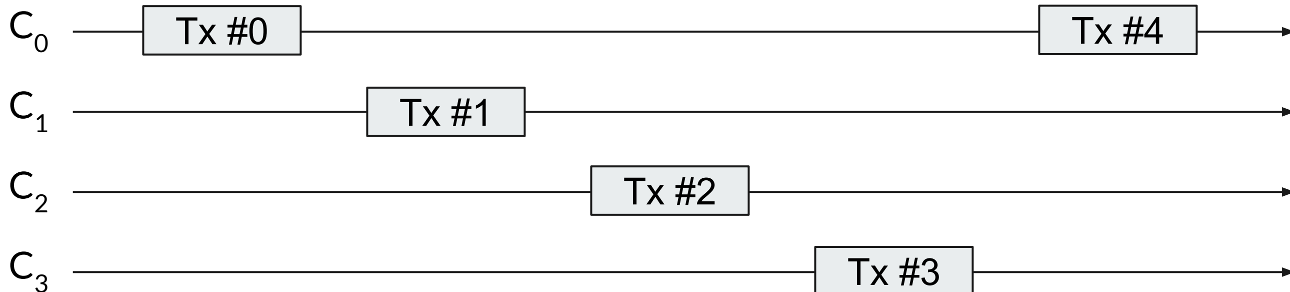
Tx B
FROM Charlie
TO Bob
AMOUNT 5.-CHF

Read C → 20
Read B → 20

Write B ←25
Read C →20
Write C ← 15

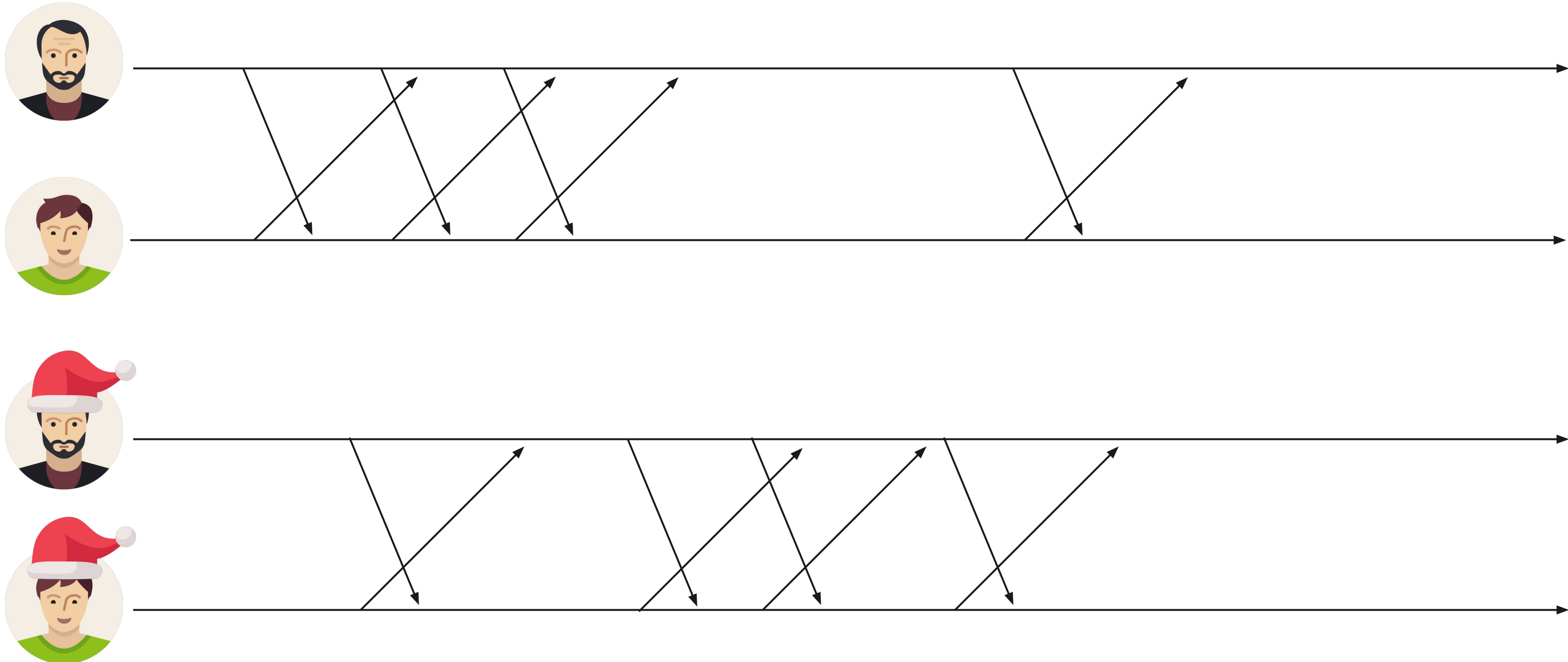
A quick fix

```
integer[] accounts;

fn safe_transfer(src, dst, amount)  {
    lock accounts { // Only one CORE can access accounts at a time
        fn transfer(src, dst, amount);
    }
}
```



Coarse-grained locking is safe, but doesn't scale.

Any parallelism?



(Santa Bob and Santa Charlie live in a different universe and don't trade with Boring Bob and Boring Charlie.)

Alice’s wish list

Alice would like an abstraction that:

- Is (almost) as simple to use as a coarse-grained lock,
- Allows non-conflicting *transactions* (transfers in our case) to run in parallel,
- Serializes conflicting transactions.

That’s exactly Transactional Memory!

Informally, a memory transaction is

- A **sequence** of actions (READs/WRITEs/ALLOCs/FREEs),
- That execute **atomically** (partially-executed transactions are not visible),
- And either **all** actions **commit** or **abort together** (all or nothing).

A Software Transactional Memory (STM) = a framework that runs memory transactions.

Safe transfers using an STM

```
// Initialization of the transactional memory, which holds the accounts
tm = new TransactionalMemory(/*...*/);
integer[] accounts = tm.get_start();

fn transfer(src, dst, amount)  {
    while (true) { try {
        let tx = tm.begin();
        if (tx.read(&accounts[src]) < amount) {
            tx.commit();
            break;
        }
        tx.write(&accounts[dst], tx.read(&accounts[dst]) + amount);
        tx.write(&accounts[src], tx.read(&accounts[src]) - amount);
        tx.commit();
        break;
    } except (RetryTransaction) { continue; } }
}
```

Implement an STM to help Alice's business grow!

9 functions to implement:

- tm_create / tm_destroy // constructor/destructor for TM
- tx_begin / tx_end // starts/tries to commit a transaction
- tx_read // reads a value in TM (in a transaction)
- tx_write // writes a value in TM (in a transaction)
- tx_alloc // allocates a new TM buffer and returns its address (in a transaction)
- tx_free // frees a previously allocated TM buffer (in a transaction)
- tm_start // returns the address of the start of the TM

Correctness of your implementation

Your STM should make concurrent transactions appear *as if* they were serial.

Must have the 3 following properties:

- Snapshot isolation: Transactions see (via reads) a snapshot of the TM, except for their own writes.
- Atomicity: All modifications from a transaction appear to take place at one indivisible point in time.
- Consistency: Transactions see the state left by the latest committed transaction (according to the *linearization* order).

Resources

- *The* project reference document (19-page pdf) on Moodle:
  - Everything in this presentation,
  - Formal specs of all functions, properties, etc.
  - Possible implementation (with high-level pseudocode),
  - Instructions to test and submit your work.

- GitHub repo: https://github.com/LPD-EPFL/CS453-2025-project
  - include/ headers
  - template/ where to work
  - reference/ a correct but SLOW implementation that uses a global lock
  - grading/ to grade your work
  - a python script to submit your work

Grading (1/2)

- 30% of the grade of Concurrent Computing.
- Correctness is a MUST: incorrect => no passing grade for the project (< 4).
- Additional correctness workloads will be released during the semester.
- FORBIDDEN to have an implementation similar to the reference one (i.e., that uses a coarse lock).
- FORBIDDEN to **never** let non-conflicting transactions run in parallel.
- Your grade depends on the speedup vs the reference implementation.
- 8x slower than the reference version => no passing grade.
- Projects with memory leaks are capped at 5.

Speedup s	**Grade** g
$s < 1/8$	$g < 4$
$s \in [1/8, 1]$	$g = 4 + \frac{8s-7}{7}$
$s \in [1, 4]$	$g = 5 + \frac{s-1}{3}$
$s > 4$	$g = 6$

Grading (2/2)

- Automated grader with unlimited submissions (only your best speedup will be kept).

- Collaborate with other students (debugging, sharing ideas, etc.), take inspiration from existing STMs, etc., but **submit YOUR OWN implementation, LLMs are PROHIBITED**.

- Any attempt to **CHEAT** (e.g., tricking the grader, plagiarism) will be **SEVERELY PUNISHED**.

Important notes

- You can use C (default) or C++.
- Locally, use the OS+compiler combo you want, but we will only troubleshoot Ubuntu/Debian+gcc/clang.
- Libraries that solve (non-elementary) concurrency tasks are forbidden.
- You will soon receive your (secret) UUID to submit your work.

Where to go

- Read *the* pdf.
- Read it again.
- Clone the repository.
- Understand the reference (bad) implementation.
- Implement the algorithm proposed in the pdf.
- Test/grade locally until you're satisfied.
- Check that your code passes new correctness workloads.
- Submit your work: done. :)

- Few locs (~1000), but tricky ones.
- Debugging concurrent code is ***VERY*** hard, **START** as **SOON** as possible.

- Seriously, for your own good, start now.

Questions?