Transaction Processor

> *A small CLI to process transaction data.*

Implementation Details

Parallelization Model

Reading the CSV file

The file is being read using `serde` and `csv`. An iterator over owned records is passed to the transaction engine and processed lazily using asynchronous code.

Transaction Engine

Accounts are represented using an `RwLock<Vec<AccountAccessor>>`. The `AccountAccessor` contains the `client_id` and an `Arc<Mutex<Account>>`.

Finding/creating an account

A. *The account already exists*: Acquire a read-lock on the `accounts` vector to avoid blocking other threads.

B. *The account doesn't exist*: Briefly acquire an exclusive read-write-lock on the `accounts` vector and immediately release it after the account has been created.

In either case, a read-lock is acquired afterwards to access account data.

Accessing the account for the current transaction

The read-lock on `accounts` is used to obtain a mutable reference to the client `Account`. The specific account used in the transaction is mutex-locked for the remainder of function execution in order to ensure secure access to account data.

More about the locking mechanism

By combining an `RwLock` for the list of accounts with the `AccountAccessor` and an `Arc<Mutex<T>>` for the individual accounts, we can ensure that many threads can concurrently find accounts and obtain mutable references to them.

The `AccountAccessor` is used for finding an account by just read-locking the `accounts` vector without having to mutex-lock individual accounts.

Blocking only briefly occurs on the `accounts` vector if an account doesn't exist, and on the `Account` for the current transaction. This way, many transactions can be processed at once and only transactions for the same client have to wait.

Assumptions

- Handling of disputes for already disputed transactions is unspecified
  - Assumption: This is a no-op. Don't throw an error and just ignore the tx.
- Handling of locked accounts is unspecified
  - Assumption: Don't throw an error, but ignore all further transaction for the client.

Additional Notes

Sample Data Generator

For lack of better testing data, a small node tool `txgen.js` can be used.
This tool will generate a million transactions and write them to stdout.

Example usage: `node ./helpers/txgen.js > transactions.csv`

This dataset can then be parsed using the tx engine:
`cargo run --release -- transactions.csv > output.csv`

Performance with generated sample datasets:
`1,000,000` transactions (`42 MB`): `1.8s`
`10,000,000` transactions (`441 MB`): `19.4s`

Parallelism Experiments

I've experimented with using `tokio` and `rayon`. Neither tokio nor rayon added a significant benefit though, since tokio doesn't do exceptionally well on purely CPU-bound workloads and rayon had (at least in my testing) some issues with missing iterations, which caused tests to fail randomly.

I've opted to use `tokio` very sparingly to allow for future parallelization of `process_transactions`. Even though the use of `tokio` doesn't make a difference at the moment, it would allow for processing multiple transaction files at the same time with only small code changes.

Currently, transactions are processed in sequence:

```
for record in records {
  Self::process_transaction(&self.accounts, record?).unwrap()
}
```

The rayon equivalent looks like this:

```
records.par_bridge().for_each(|record| {
    if let Ok(record) = record {
        Self::process_transaction(&self.accounts, record).unwrap()
    }
});
```

Lastly, here is the same code using tokio:

```
let mut handles = Vec::new();
let mut stream = tokio_stream::iter(records);
while let Some(record) = stream.next().await {
  if let Ok(record) = record {
    let accounts = self.accounts.clone();
    let handle = tokio::spawn(async move {
      Self::process_transaction(accounts, record).await
    });
    handles.push(handle);
  }
}
for handle in handles {
```

```
  handle.await??;
}
```