Concurrent transaction Executor - public Assignment

Objective:

Your task is to implement a concurrent transaction executor for a blockchain. The executor processes transactions within a block concurrently, ensuring the global account state is updated consistently and safely.

Background:

Accounts represent the state of the blockchain. Each account has two properties:

- A **name** (a string) which uniquely identifies the account.
- A **balance** (an unsigned integer) representing the account's balance or state.

Transaction is a function that takes the current account state and returns a list of updates to the accounts. Each update is a key/value pair, where the key is the account name, and the value is the balance change for that account.

Block is an ordered sequence of transactions. If a transaction in a block fails other transactions can still be run.

Account State is the snapshot of accounts at a given block.

Task Overview:

The input to the system will a series of `Block` objects, which are defined using the following data structures (sample code in Go):

```
type Block struct {
    Transactions []Transaction
}

type Transaction interface {
    Updates(AccountState) ([]AccountUpdate, error)
}
```

```
// Updates
type AccountUpdate struct {
    Name string
    BalanceChange int
}

type AccountValue struct {
    Name string
    Balance uint
}

// if the account does not exist, return zero balance
type AccountState interface {
    GetAccount(name string) AccountValue
}

// ExecuteBlock takes a Block with transactions, and returns
the updated account and with the updated balance.
func ExecuteBlock(Block) ([]AccountValue, error) {
    // Implement this
}
```

The solution should use N workers to deterministically process all transactions within the block as efficiently as possible. It must handle cases where transactions within the same block modify overlapping accounts.

The number of workers N will be statically defined as constant, but should be tuneable.

After executing all transactions in the block, the final global account state should reflect all committed changes.

Assume that the entire AccountState is very large.

This is an example of a transfer transaction that transfers some value from `from` account to `to` account:

```
type transfer struct {
    from string
    to string
    value int
}

func (t transfer) Updates(state AccountState) ([]AccountUpdat
e, error) {
     fromAcc := state.GetAccount(t.from)
     if fromAcc.Balance < t.value {
         return nil, fmt.Error("")
     }

     return []AccountUpdate{
         {Name: t.from, BalanceChange: -t.value},
         {Name: t.to, BalanceChange: t.to}
     }, nil
}
```

Example 1

As an example, given the current account state has three accounts: A, B, C.

A has a Balance 20
B has a Balance 30
C has a Balance 40

```
[
    AccountValue{ Name: "A", Balance: 20 },
    AccountValue{ Name: "B", Balance: 30 },
    AccountValue{ Name: "C", Balance: 40 },
]
```

Now a block has 3 transactions:
The first transaction is A transfer 5 tokens to B. So it will return account updates as:

```
[
    AccountUpdate{ Name: "A", BalanceChange: -5 },
    AccountUpdate{ Name: "B", BalanceChange: 5 },
]
```

The second transaction is B transfer 10 tokens to C. So it will return account updates as:

```
[
    AccountUpdate{ Name: "B", BalanceChange: -10 },
    AccountUpdate{ Name: "C", BalanceChange: 10 },
]
```

The third transaction is B transfer 30 tokens to C. However the transaction fails because B's balance is less then 30.

```
[
  // no updates to apply
]
```

Executing this block will return a list of updated account value:

```
[
    AccountValue{ Name: "A", Balance: 15 },
    AccountValue{ Name: "B", Balance: 25 },
    AccountValue{ Name: "C", Balance: 50 },
]
```

Example 2

As an example, given the current account state has three accounts: A, B, C.

A has a Balance 10

B has a Balance 20

C has a Balance 30

D has a Balance 40

```
[
    AccountValue{ Name: "A", Balance: 10 },
    AccountValue{ Name: "B", Balance: 20 },
    AccountValue{ Name: "C", Balance: 30 },
    AccountValue{ Name: "D", Balance: 40 },
]
```

Now a block has 2 transactions:
The first transaction is A transfer 5 tokens to B. So it will return account updates as:

```
[
    AccountUpdate{ Name: "A", BalanceChange: -5 },
    AccountUpdate{ Name: "B", BalanceChange: 5 },
]
```

The second transaction is C transfer 10 tokens back to D. So it will return account updates as:

```
[
    AccountUpdate{ Name: "C", BalanceChange: -10 },
    AccountUpdate{ Name: "D", BalanceChange: 10 },
]
```

Executing this block will return a list of updated account value:

```
[
    AccountValue{ Name: "A", Balance: 5 },
    AccountValue{ Name: "B", Balance: 25 },
    AccountValue{ Name: "C", Balance: 20 },
```

```
    AccountValue{ Name: "D", Balance: 50 },
]
```

Since the two transactions are touching different accounts that have no overlap, the 2 transactions can be run concurrently and still produce the same `[]AccountValue` as updates.

Requirements:

- The result of executing a block must be deterministic. i.e. it must be reproducible by any node running the same software with the same set of inputs. Said another way, the system should produce the same result with 1 worker as with an arbitrarily large number of workers.
- The solution must be efficient and capable of scaling with different values of N, as well as block, transaction, and state sizes.