A Concurrent Perspective on Smart Contracts

Flavius Andrei Isac
04.05.2018
Summary

- Introduction
- Concurrency
- Smart Contracts
- Examples of Deployed "Concurrent" Smart Contracts
  - The BlockKing Contract
  - The DAO Contract
  - Personal Example?
- Discussion of Other Possible Issues
  - Interference and Synchronization
  - State Ownership and Permission Accounting
- Conclusion
Smart Contracts

- Programs stores on the blockchain
- Triggered by transactions
- Read and write data on the blockchain

- Sequential Programmes
  - Reentrancy
  - Recursive calls

```solidity
contract token {
  mapping (addr public coinBalanceOf:
  event CoinTransfer sender, address reci;
  function transfer (uint if supply (sup
  10000; coinBalanceOf[ supply;
  }
  signature 1
  signature 2
```
We will look at the properties of their concurrent executions

Execution can span during multiple transactions

- Within the same block
- Within multiple blocks

Violates the atomicity property

State of the contract can be changed by other parties
Concurrency

- Concurrent objects
  - Locks
  - Queues
  - Atomic counters

- Multiple threads accessing same object => **Inference**

- **Race conditions** => loss of memory integrity

- Synchronisation
Smart Contracts vs Concurrent Objects (blockchain vs shared memory)

- Accounts using smart contracts in a blockchain are like threads using concurrent objects in shared memory.

- Both have internal mutable state (e.g. funds)

- Can be accessed by multiple parties
Smart Contracts vs Concurrent Objects
(blockchain vs shared memory)

- Used by accounts (users or other contracts)
- A call to a contract is executed sequentially without interrupts
- Method calls are atomic, either successfully updates the blockchain or rolls back
- Used by threads

Differences
Smart Contracts – not exactly fully atomic

- Order of transactions specified at the moment of transaction execution
- Transactions can be spread across several blocks (e.g. Oraclize)
- Calling other contracts – cooperative multitasking – yield – the DAO Bug – expand later
- Contract as a service (e.g. managing a shared resource) – account for inference patterns
Deployed ”Concurrent” Smart Contracts

1. BlockKing
   - Gambling game
   - Oraclize

2. DAO
   - “Uncooperative multitasking”
   - Reentrancy
function enter() {
    // 100 finney = .05 ether minimum payment otherwise refund payment and stop contract
    if (msg.value < 50 finney) {
        msg.sender.send(msg.value);
        return;
    }
    warrior = msg.sender;
    warriorGold = msg.value;
    warriorBlock = block.number;
    bytes32 myid =
        oraclize_query(0,"WolframAlpha","random number between 1 and 9");
}

function __callback(bytes32 myid, string result) {
    if (msg.sender != oraclize_cbAddress()) throw;
    randomNumber = uint(bytes(result)[0]) - 48;
    process_payment();
}

function process_payment() {
```solidity
function enter() {
    // 100 finney = 0.05 ether minimum payment otherwise refund payment and stop contract
    if (msg.value < 50 finney) {
        msg.sender.send(msg.value);
        return;
    }
    warrior = msg.sender;
    warriorGold = msg.value;
    warriorBlock = block.number;
    bytes32 myid =
        oraclize_query(0,"WolframAlpha","random number between 1 and 9");
}

function __callback(bytes32 myid, string result) {
    if (msg.sender != oraclize_cbAddress()) throw;
    randomNumber = uint(bytes(result)[0]) - 48;
    process_payment();
}

function process_payment() {
```
function enter() {
    // 100 finney = .05 ether minimum payment otherwise refund payment and stop contract
    if (msg.value < 50 finney) {
        msg.sender.send(msg.value);
        return;
    }
    warrior = msg.sender;
    warriorGold = msg.value;
    warriorBlock = block.number;
    bytes32 myid =
        oraclize_query(0, "WolframAlpha", "random number between 1 and 9");
}

function __callback(bytes32 myid, string result) {
    if (msg.sender != oraclize_cbAddress()) throw;
    randomNumber = uint(bytes(result)[0]) - 48;
    process_payment();
}

function process_payment() {

2 - querying Oraclize

BlockKing

Block N

Transaction 1
- c.prepareRequest()
- o.raiseEvent()

Block N+M

Transaction 2
- o.respond()
- c.__callback(data)
function enter() {
    // 100 finney = .05 ether minimum payment otherwise refund payment and stop contract
    if (msg.value < 50 finney) {
        msg.sender.send(msg.value);
        return;
    }
    warrior = msg.sender;
    warriorGold = msg.value;
    warriorBlock = block.number;
    bytes32 myid =
        oraclize_query(0,"WolframAlpha","random number between 1 and 9");
}

function __callback(bytes32 myid, string result) {
    if (msg.sender != oraclize_cbAddress()) throw;
    randomNumber = uint(bytes(result)[0]) - 48;
    process_payment();
}

function process_payment() {
function process_payment() {

...

if (singleDigitBlock == randomNumber) {
    rewardPercent = 50;
    // If the payment was more than .999 ether then increase reward percentage
    if (warriorGold > 999 finney) {
        rewardPercent = 75;
    }
    king = warrior;
    kingBlock = warriorBlock;
}
DAO

- Problem Idea

- First Sends the Ether, then updates the balance

- The send method “yields” control to the third party (user or contract)

- The unfair party can request again a withdrawal
DAO

1010 // Burn DAO Tokens
1011 Transfer(msg.sender, 0, balances[msg.sender]);
1012 withdrawRewardFor(msg.sender); // be nice, and get his rewards
1013 totalSupply -= balances[msg.sender];
1014 balances[msg.sender] = 0;
1015 paidOut[msg.sender] = 0;
1016 return true;
1017 }
// Burn DAO Tokens
Transfer(msg.sender, 0, balances[msg.sender]);
withdrawRewardFor(msg.sender); // be nice, and get his rewards
totalSupply -= balances[msg.sender];
balances[msg.sender] = 0;
paidOut[msg.sender] = 0;
return true;
DAO – rewrite example

User contract

```solidity
function foo() {
    wallet.withdraw();
}

function () payable {
    wallet.withdraw();
}
```

Wallet contract

```solidity
uint balance = 10;

function withdraw() {
    if (balance > 0)
        msg.sender.call.value(balance)();
    balance = 0;
}
```

By default, i.e. using `call.value(amountEther)()`, sends all remaining gas to the receiver

*Thanks to the PASS Course*
Interference and Synchronization
Atomic updates in shared-memory concurrency

class Counter {
    private int x = 0;

    /** Return current value */
    synchronized int get() {
        return x;
    }

    /** Set x to be v */
    synchronized int set(int v) {
        int t = x;
        x = v;
        return t;
    }
}

final Counter c = new Counter();

void incr() {
    int a = c.get();
    int b = c.set(a + 1);
    assert (a == b);
}

// In the main method
Runnable thread1 = () -> {
    incr();
}

Runnable thread2 = () -> {
    incr();
}

thread1.run(); thread2.run();
Interference and Synchronization
Atomic updates in shared-memory concurrency

class Counter {
    private int x = 0;

    /** Return current value */
    synchronized int get() {
        return x;
    }

    /** Set x to be v */
    synchronized int set(int v) {
        int t = x;
        x = v;
        return t;
    }
}

final Counter c = new Counter();

void incr() {
    int a = c.get();
    int b = c.set(a + 1);
    assert (a == b);
}

// In the main method
Runnable thread1 = () ->
    { incr(); }

Runnable thread2 = () ->
    { incr(); }

thread1.run(); thread2.run();
Interference and Synchronization
Atomic updates in concurrent blockchain transactions

```solidity
contract Counter {
    address public id;
    uint private balance;

    function get() returns (uint) {
        return balance;
    }

    function set() returns (uint) {
        uint t = balance;
        balance = msg.value;
        msg.sender.send(t);
        return t;
    }
}

// ...
// Same code as in Counter

function testAndSet(uint expected)
returns (uint) {
    uint t = balance;
    if (t == expected) {
        balance = msg.value;
        msg.sender.send(t);
        return t;
    } else {
        throw;
    }
}
```
Interference and Synchronization
Atomic updates in concurrent blockchain transactions

- Multiple consecutive get and set calls will end up in two different transactions, launched by multiple parties trying to modify the Counter at the same time.

Interference and Synchronization
Atomic updates in concurrent blockchain transactions

```solidity
function testAndSet(uint expected) {
    uint t = balance;
    if (t == expected) {
        balance = msg.value;
        msg.sender.send(t);
        return t;
    } else {
        throw;
    }
}
```

```solidity
function cas(p : pointer to int, old : int, new : int) returns bool {
    if *p != old {
        return false
    }
    *p ← new
    return true
}
```
Interference and Synchronization

Atomic updates in concurrent blockchain transactions

```solidity
function testAndSet(uint expected)
    returns (uint) {
    uint t = balance;
    if (t == expected) {
        balance = msg.value;
        msg.sender.send(t);
        return t;
    } else {
        throw;
    }
}
```

```solidity
function cas(p : pointer to int, old : int, new : int) returns bool {
    if (*p != old) {
        return false
    }
    *p = new
    return true
}
```
State Ownership and Permission Accounting

- State ownership discipline enforced by modifiers in Solidity

- Synchronisation can be obtained by implementing a basic Read/Write lock inside the contract

- **Issue**: Implementation design is non-modular
contract Counter {
    address public owner;
    uint private balance;

    modifier byOwner() {
        if (msg.sender != owner) throw;
    }

    function get() external byOwner
        returns (uint) {
        return balance;
    }

    function set() external byOwner
        returns (uint) {
        uint t = balance;
        balance = msg.value;
        msg.sender.send(t);
        return t;
    }

    mapping (address => bool) readers;

    // Initialized with 0x0
    address writer;

    modifier canRead() {
        if (msg.sender != writer ||
            !readers[msg.sender]) throw;
    }

    modifier canWrite() {
        if (msg.sender != writer) throw;
    }

    function acquireReadLock() returns (bool) {
        if (writer == 0x0) {
            readers[msg.sender] = true;
        } else return false;
    }
State Ownership and Permission Accounting

```solidity
contract Counter {
    address public owner;
    uint private balance;

    modifier byOwner() {
        if (msg.sender != owner) throw;
    }

    function get() external byOwner returns (uint) {
        return balance;
    }

    function set() external byOwner returns (uint) {
        uint t = balance;
        balance = msg.value;
        msg.sender.send(t);
        return t;
    }
}

mapping (address => bool) readers;

// Initialized with 0x0
address writer;

modifier canRead() {
    if (msg.sender != writer || !readers[msg.sender]) throw;
}

modifier canWrite() {
    if (msg.sender != writer) throw;
}

function acquireReadLock() returns (bool) {
    if (writer == 0x0) {
        readers[msg.sender] = true;
    } else return false;
}
```
contract Counter {
    address public owner;
    uint private balance;

    modifier byOwner() {
        if (msg.sender != owner) throw;
    }

    function get() external byOwner
        returns (uint) {
        return balance;
    }

    function set() external byOwner
        returns (uint) {
        uint t = balance;
        balance = msg.value;
        msg.sender.send(t);
        return t;
    }

    mapping (address => bool) readers;

    // Initialized with 0x0
    address writer;

    modifier canRead() {
        if (msg.sender != writer || !readers[msg.sender]) throw;
    }

    modifier canWrite() {
        if (msg.sender != writer) throw;
    }

    function acquireReadLock() returns (bool) {
        if (writer == 0x0) {
            readers[msg.sender] = true;
        } else return false;
    }
contract Counter {
    address public owner;
    uint private balance;

    modifier byOwner() {
        if (msg.sender != owner) throw;
    }

    function get() external byOwner returns (uint) {
        return balance;
    }

    function set() external byOwner returns (uint) {
        uint t = balance;
        balance = msg.value;
        msg.sender.send(t);
        return t;
    }
}

mapping (address => bool) readers;

// Initialized with 0x0
address writer;

modifier canRead() {
    if (msg.sender != writer || !readers[msg.sender]) throw;
}

modifier canWrite() {
    if (msg.sender != writer) throw;
}

function acquireReadLock() returns (bool) {
    if (writer == 0x0) {
        readers[msg.sender] = true;
    } else return false;
}
State Ownership and Permission Accounting

```solidity
contract Counter {
    address public owner;
    uint private balance;

    modifier byOwner() {
        if (msg.sender != owner) throw;
    }

    function get() external byOwner returns (uint) {
        return balance;
    }

    function set() external byOwner returns (uint) {
        uint t = balance;
        balance = msg.value;
        msg.sender.send(t);
        return t;
    }

    mapping (address => bool) readers;

    // Initialized with 0x0
    address writer;

    modifier canRead() {
        if (msg.sender != writer || !readers[msg.sender]) throw;
    }

    modifier canWrite() {
        if (msg.sender != writer) throw;
    }

    function acquireReadLock() returns (bool) {
        if (writer == 0x0) {
            readers[msg.sender] = true;
        } else return false;
    }
}
```
State Ownership and Permission Accounting

```solidity
contract Counter {
    address public owner;
    uint private balance;

    modifier byOwner() {
        if (msg.sender != owner) throw;
        
    }

    function get() external byOwner returns (uint) {
        return balance;
    }

    function set() external byOwner returns (uint) {
        uint t = balance;
        balance = msg.value;
        msg.sender.send(t);
        return t;
    }

    mapping (address => bool) readers;

    // Initialized with 0x0
    address writer;

    modifier canRead() {
        if (msg.sender != writer || !readers[msg.sender]) throw;
    }

    modifier canWrite() {
        if (msg.sender != writer) throw;
    }

    function acquireReadLock() returns (bool) {
        if (writer == 0x0) {
            readers[msg.sender] = true;
            return true;
        } else return false;
    }
}
```
Discussion
Composing the contracts

- Non-modularity of the Read/Write lock
- SWE principles -> separation of concerns
- Synchronisation primitives -> standalone libraries

However:
- Difficult to reason about multi-contract interaction behaviour
- Libraries exists, but without any internal state
- “contract factory” – not safe
Discussion

Liveness properties

- Liveness = “eventually something good happens”
- Fairness of the system scheduler
Conclusions

- New perspective: smart contract ~ concurrent object

- Take into account that not all concurrency problems have a direct counterpart in smart contracts

- Understanding *intra-* and *inter-*transactional behaviour

- Detecting *atomicity violations* and *data races*
Accounts using **smart contracts** in a blockchain are like *threads* using **concurrent objects** in shared memory.