A gentle introduction to the Blockchain and Smart contracts

Giovanni Ciatto  { giovanni.ciatto@unibo.it }

Talk @ Autonomous Systems Course, A.Y. 17/18
Dipartimento di Informatica, Scienza e Ingegneria—Università di Bologna

May 30, 2018
I wish to thank my supervisor Prof. Andrea Omicini, and my colleagues Prof. Enrico Denti, Dr. Stefano Mariani, and Dr. Roberta Calegari for the many fruitful discussions which I tried to synthesise in these slides.

— G. Ciatto
Talk Outline

1. State Machine Replication
2. The blockchain’s main elements
3. Smart contracts
4. Research perspectives
State Machine Replication (SMR) [24, 10]

Main idea

Executing the same (not necessarily finite) state machine over multiple independent (possibly distributed) processors, in parallel, in order to achieve:

- fault tolerance (to stops, crashes, lies, bugs, etc)
- availability and reactivity
- data / software replication & untamperability

\( a \) State machine \( \approx \) program

A network of replicas

When distributed, we say the processors constitute a peer-to-peer (P2P) network of replicas, all exposing the same observable behaviour

\(!\) no assumption about the topology
State Machine Replication (SMR)
Deterministic stateless computations

**Input** → **Computation** → **Output**

The computation is deterministic if it *always* produces the same output when it is performed on the same input.

- Can be arbitrarily replicated
- Replicas can be run in parallel

```java
class Calculator {
    int sum(int x, int y) {
        return x + y;
    }
}
```

```java
class Lottery {
    int extract(int max) {
        Random fate = new Random();
        return fate.nextInt(max);
    }
}
```
Deterministic stateful computations

\[(\text{Input, State}) \rightarrow \text{Computation} \rightarrow (\text{Output, State}')\]

The computation is deterministic if it \textit{always} produces the same output when it is performed on the same input \textbf{and} state.

- Can be arbitrarily replicated & replicas run in parallel
- All replicas must be initialised within the same initial state
- All inputs must be submitted to all replicas in the same order\(^1\)
  - They all move through the same sequence of states
  - Maintaining the consistency of the state among on inputs

\(^1\)input \approx method call
Deterministic stateful computations II

```java
class Ledger {
    Map<String, Integer> balances = // all accounts to 0

    void deposit(String userID, int money) {
        balances[userID] += money;
    }

    boolean transfer(String sender, String receiver, int money) {
        if (balances[sender] >= money) {
            balances[sender] -= money;
            balances[receiver] += money;
            return true;
        }
        return false;
    }
}
```
Non-deterministic

```java
class RaceCondition {
    int shared = 0;
    Thread t1 = new Thread(() -> shared++);
    Thread t2 = new Thread(() -> shared--);

    int guess() {
        t1.start(); t2.start(); t1.join(); t2.join();
        return shared;
    }
}
```

The Blockchain is essentially a means for replicating deterministic stateful computations
We can replicate a stateful deterministic program implementing a particular business logic

- In the exact same way, we could replicate a deterministic program implementing an interpreter

- The API of such an interpreter would enable programs to be deployed, undeployed, invoked, etc.

Smart-contracts-enabled Blockchains essentially act as replicated “universal” state machines on which smart contracts (SC) can be deployed
```
class VirtualMachine {
    Map<String, Program> processes = // empty
    Compiler cc = // ...

    void deploy(String pid, String code) {
        Program newProgram = cc.compile(code);
        processes[pid] += newProgram;
    }

    Object invoke(String pid, Object[] args) {
        Object result = null;
        if (processes.containsKey(pid)) {
            result = processes[pid].call(args);
        }
        return result;
    }
}
```
SMR and **Distributed Systems**

- Messages may be lost, reordered, or duplicated by the network\(^2\):
  - each node may **perceive** a different view about the system events

- Lack of global time
  - \(\implies\) lack of total ordering of events
  - \(\implies\) lack of trivial consistency

- **Consistency, Availability, Partition-resistance (CAP) theorem** \([7]\)
  - \(\implies\) you **cannot** have more than **two** of them

- Authentication is required if the replicated service is user-specific

\(^2\)messages \(\approx\) inputs to replicated processes
SMR, Middleware, and Consensus I

Each replica is executed on top of a middleware taking care of validating & ordering inputs for the replicated program.

- It is then invoked on all nodes with the same sequence of inputs.
- The middleware makes nodes participate to a consensus protocol:
  - i.e. a distributed algorithm aimed at selecting the next input
  - ... producing the so-called atomic broadcast

\[ \text{Fischer, Lynch and Patterson (FLP) theorem } [15] \]
\[ \implies \text{impossibility of consensus without timing assumptions} \]
SMR, Middleware, and Consensus II
SMR and Open Distributed Systems

- How can we prevent a protocol participant from
  - lying w.r.t. the protocol rules or exchanged data?
  - being buggy, therefore breaking the rules or producing wrong data?
  - crashing?
  - ... in general: being byzantine?

- Long story short: we can’t.
- BUT we can tolerate some byzantine nodes
  - Less than 1/3 of the total amount of nodes, according to
    Lamport’s Byzantine Generals Problem solution [19]

- We can also ease the recognition of prohibited or unauthorised
  behaviours by employing cryptography
  - e.g. Pub/Priv key pairs for user authentications
  - e.g. 1-Way Hash functions and MAC for data integrity
SMR and Open Distributed Systems

Takeaway

The blockchain is a smart way to achieve (U)SMR, dealing with – i.e., mitigating – well known issues of open distributed systems.
Most of Blockchain-related works describe a specific blockchain technology (BCT henceforth) using a bottom-up approach. I believe this approach hinders generality and limits the discussion about what we can do on top of BCTs. In this section, I try to present the blockchain in a top-down way, synthesising informations from a number of sources, being [23], [28], [3] the most prominent ones. Errors and misunderstanding are possible, and in any case they are my sole responsibility.

The following description of the blockchain architecture and functioning is strongly inspired to Ethereum$^3$, being the most mature, studied, and documented smart-contracts-enabled BCT.

— G. Ciatto

$^3$https://github.com/ethereum/wiki/wiki/White-Paper
Overview

Blockchain Technology (BCT)
A clever implementation of a SMR system keeping track of which users own some assets (representations), by means of a replicated ledger
e.g. The Ledger snippet

Smart-contracts-enabled BCT
A clever implementation of a USMR system keeping track of assets (representations) owned by entities – there including smart-contracts (SC), i.e. processes, owning code and state –, by means of a replicated ledger
e.g. The VirtualMachine snippet
Entity identifiers

Users

- Users are supposed to own (at least) one \((K_{pub}, K_{pr})\) key pair
- They are identified by some function \(f(K_{pub})\) of their public key
  - e.g. 1-way-hash functions
  - e.g. digital certificates issued by some trusted CA

- Identifiers are also known as addresses in this context

Permissioned vs Permissionless

- Either each user owns multiple non-intelligible identifiers...
  - ✓ Pseudonymity
  - ✓ Decentralised
  - ✗ Sybil-attack [12]
- ...or he/she owns a single certified identifier
  - ✗ Single point of failure/trust

- Smart-contracts-enabled BCTs identify both smart contracts’ instances and users by means of the same sort of addresses
The system state

\[
\langle SystemState \rangle ::= entityID \mapsto \langle Account \rangle \\
\lor \langle SystemState \rangle \langle SystemState \rangle
\]

The system state *conceptually* consists of a mapping between entity identifiers and *arbitrary* account data related to that account.
The world state II

The account state

\[ \langle Account \rangle ::= (\text{balance}, \langle Storage \rangle, \langle Code \rangle, \langle Metadata \rangle) \]

The account state *conceptually* consists of several fields keeping track of what a particular entity currently owns. The fields may vary depending on

- The blockchain nature
- Whether the entity is a smart contract or a user

*e.g.* BCTs coming with native cryptocurrencies, usually keep track of accounts *balances* (at least)

*e.g.* Smart-contracts-enabled BCTs, may keep track of their source code and internal \( \langle Storage \rangle \)
The world state III

```java
class Ledger {
    Map<String, Integer> balances = ...
    //  ^^^^^^^^  // system state

    void deposit(String userID, int money) {
        //  ^^^^^^  // entity identifier
        balances[userID] += ...
        //  ^^^^^^^^^^^  // account state
    }
}
```

Several possible implementations

- **Unspent Transaction Output (UTXO)**
  - e.g. Bitcoin [23]

- **Account-based**
  - e.g. Ethereum [28]

- **Key-value store**
  - e.g. Hyperledger Fabric [3]
Transactions (a.k.a. inputs or messages) I

(Informal) Definition

\[ \langle Transaction \rangle ::= (txID, issuerID, \langle Signature \rangle, recipientID, value, \langle Data \rangle) \]

Transactions encode (world) **state variations** yet to be performed.

- **txID** — the transaction progressive number
- **issuerID** — the address of the transaction issuer entity
- **\langle Signature \rangle** — the cryptographic signature of the transaction
- **recipientID** — the address of the transaction recipient entity
- **value** — some non negative amount of cryptocurrency
- **\langle Data \rangle** — some arbitrary data
Transactions (a.k.a. inputs or messages) II

Transaction use cases (for smart-contracts-enabled BCTs)

- **Deployment TX** — if $\text{recipientID} = \emptyset \land \langle Data \rangle = \text{code}$
- **Invocation TX** — if $\text{recipientID} = \text{address} \land \langle Data \rangle = \text{whatever}$
- **Money transfer TX** — if $\text{recipientID} \neq \emptyset \land \text{value} > 0 \land \langle Data \rangle = \varepsilon$

Transactions life-cycle — part 1

1. A issuer user **compiles** a transaction
2. He/she **signs** it with his/her private key $K_{pr}$
3. His/her node **spreads** the transaction over the P2P network
4. Peers only take into account **valid** transactions
5. ...
Transactions (a.k.a. inputs or messages) III

Transactions validity

In order for a transaction to avoid being dropped by peers:

- it must be well formed
- the signature must match the issuer address
- the signature must certify the transaction integrity
- the issuer’s balance must be $\geq$ value

! Even once a valid transaction has been spreaded over the network, there is no guarantee on when it will be executed
Blocks and block chains I

(Informal) Definition

\[ \langle \text{Block} \rangle ::= (\text{prevHash}, \text{index}, \text{time}, \langle \text{TxList} \rangle, \langle \text{FinalState} \rangle) \]
\[ \langle \text{TxList} \rangle ::= (\langle \text{Transaction} \rangle, \langle \text{IntermediateState} \rangle) \]
\[ | \langle \text{TxList} \rangle \langle \text{TxList} \rangle \]

Blocks are timestamped, hash-linked lists of transactions.

prevHash — the hash of the previous block
index — the index of the current block
time — the timestamp of the current block
\[ \langle \text{TxList} \rangle \] — the list of transactions included into the current block and the intermediate system states they produces
\[ \langle \text{FinalState} \rangle \] — the system state resulting from applying all transactions in \[ \langle \text{TxList} \rangle \], respecting their order
Blocks and block chains II

Figure: Graphical representation of the Block-chain from the global p.o.v.
Blocks features

- Replication + Hash-chaining $\leadsto$ Untamperability of the past

- Hash-chain + Time + Ordering $\leadsto$ Timestamping/notary service \[16\]

- Hash-chain + Crypt. signatures $\leadsto$ \{Accountability, Non-repudiation\}

- They are supposed to be published (almost) periodically

! In the general case $\lim_{n \to \infty} P[inconsistent(B_i)] = 0$, where
  - $B_i$ is the $i^{th}$ block
  - $n$ is the amount of successor blocks of $B_i$
  - $inconsistent(B_i)$ is true if not all nodes agree on the content of $B_i$
A block’s life I

The **genesis** block

The very first block is assumed to be shared between the initial nodes

**Blocks life-cycle — part 1**

Each node, *periodically*:

1. listens for transactions published by other nodes
2. validates, consistency-checks & executes them
3. compiles the new local candidate block
4. participates to the **consensus algorithm**

   i.e. negotiates the next block to be appended to the blockchain

   ! this phase include a spreading of the block to peers
A block’s life II

Transactions life-cycle — part 2

4. the transaction is validated by peers upon reception
   - and dropped if invalid

5. each transaction is eventually executed
   - producing an intermediary state

6. they are both included into some block

7. the block is eventually appended to the blockchain
   - i.e. a consensus protocol confirms the block
     - (there including its transactions)

! These life-cycles may vary a lot depending on the specific consensus algorithm employed
The network point of view

Figure: Graphical representation of the Block-chain from the network p.o.v.
The blockchain's main elements

Consensus & Mining

**Permission-ed BCTs**

Constrain users’ IDs through **CAs**

⇓

“Classical” quorum/leader-based consensus algorithms

- BFT algorithms
  - e.g. PBFT [9]
  - e.g. BFT-SMaRt [26]
  - e.g. HoneyBadger BFT [22]
- Non-BFT algorithms
  - e.g. Paxos [17]
  - e.g. Raft [18]
  - e.g. ZooKeeper, Google Chubby

**Permission-less BCTs**

Open access to any \((K_{pub}, K_{pr})\)

⇓

“Novel” competition-based approaches

- e.g. Proof-of-Work [6]
- e.g. Proof-of-Stake [1]
- e.g. Proof-of-Elapsed-Time [11]
- e.g. IOTA Tangle [2]

Comparisons & surveys in [5, 8]
Consensus & Mining II

**Permission-ed BCTs**

“Classical” quorum/leader-based consensus algorithms

- Assumptions on the amount $N$ of nodes
  - $UB$ up to $\sim 100 / 1000$
- High throughput in terms of TXs/second
  - $OoM \sim 1000$ TXs/s
- “Exact” consistency
- Ideal for closed multi-administrative organizations

**Permission-less BCTs**

“Novel” competition-based approaches

- No assumption on $N$
  - $UB$ virtually $\infty$
- Low throughput
  - $OoM \sim 10$ TXs/s
- Probabilistic consistency
- Ideal for open systems
Proof-of-Work (PoW)

PoW (a.k.a. mining) — the typical approach in cryptocurrencies

- Nodes (a.k.a. miners) compete to be the first one to solve a computational puzzle, once every $\Delta T$ seconds
  - finding a block hash having a given amount of leading zeros
  - hashing (pseudo)random pieces of data attained from the block content
- The proof of the effort is easy to verify and included into the block
- The block is spreaded on the P2P network
- Other miners confirm the novel block by mining on top of it
- Forks (i.e. inconsistency) are eventually aborted
  - Longest cumulative difficulty
  - Greedy Heaviest Observed SubTree (GHOST [25])
PoW interesting features

- **Competition-based, local, eventually-consistent, stochastic approach**

- **Self-adaptive mining difficulty**, s.t. $E[\Delta T] = const$
  - the system update frequency is $1/E[\Delta T]$

- **Only computing power (CP) matters here**
  - Sybil-attack resistant
  - CP distribution & Majority rule (51% attack) [14]

! Endows the cryptocurrency with its **economical value**

! Miners require **economical compensation** for their effort
The blockchain's main elements

Consensus & Mining

PoW security

\[
    r = \frac{\text{adversary}_{CP}}{\text{honest}_{CP}}
\]

\[
    \mathbb{P}[n \mid r] = 1 - \sum_{k=0}^{n} \frac{(nr)^k e^{-nr}}{k!} (1 - r^{n-k})
\]

(see [23])

In Bitcoin:

- \( n_{\text{threshold}} = 6 \)
- \( \approx 1h \) since \( \mathbb{E}[\Delta T] = 10m \)
- 99.999% secure if \( \text{adversary}_{CP} < 13\% \text{ total}_{CP} \)
## (Informal) Definition

Stateful, reactive, user-defined, immutable, and deterministic processes executing some arbitrary computation on the blockchain, i.e., while being replicated over the blockchain network.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateful</td>
<td>They encapsulate their own state, like OOP’s objects</td>
</tr>
<tr>
<td>Reactive</td>
<td>They can only be triggered by issuing some invocation TX</td>
</tr>
<tr>
<td>User-defined</td>
<td>Users can deploy their smart contracts implementing an arbitrary logic by issuing a deployment TX</td>
</tr>
<tr>
<td>Immutable</td>
<td>Their source/byte-code cannot be altered after deployment</td>
</tr>
<tr>
<td>Arbitrary</td>
<td>They are expressed with a Turing-complete language</td>
</tr>
<tr>
<td>Replicated</td>
<td>The blockchain is essentially a replicated interpreter, employing a consensus protocol to synchronise the many smart contracts replica</td>
</tr>
</tbody>
</table>
Smart contracts interesting features and expectations

- Immutability + Untamperable + Accountability + Decentralisation
  ➔ can be trusted in handling financial operations between organizations
  (even easier with native cryptocurrency)

- The code is always right, the true history is on the blockchain
  - reducing disputes
  - removing the need for arbitration

  - Like the cloud, but with no single point of control

- Killer applications: cryptocurrencies, asset-tracking (e.g. property, notary, medical-records, etc.), naming systems, ID management, access control, voting systems, reputation systems, blackboard systems, Distributed Autonomous Organizations (DAOs), etc.
Smart contracts deployment

1. Initially there exists no smart contract
   - i.e., the system state comprehend no smart contract entity

2. A user can instantiate a smart contract by publishing its source/byte-code within a deployment TX
   - the TX also initialises the SC state
   - the protocol assigns an univocal address to the novel SC

3. Once the transaction is confirmed, you can assume the SC instance is running on all nodes
   - no such a big effort: it is just listening for invocations
A user can trigger an already deployed SC by publishing an invocation TX
- specifying the SC address as recipient
- providing some input data specifying the requested computation

Eventually, each node will receive the TX and execute it
- the SC code usually decides what to do given the provided input data

If the computation terminates without exceptions, any produced side effects (on the SC state) become part of the new intermediate system state
- Otherwise, they are simply dropped, this the new intermediate state coincides with the previous one

The wrapping block is eventually confirmed by the consensus protocol, and the invoked computation can be actually considered executed
Issues arising from Turing-completeness

What would be the effect of invoking such a smart-contract?

```java
class Bomb {
    int i = 0;

    void doSomething() {
        while (true) {
            i = (i + 1) % 10;
        }
    }
}
```

- BTCs cannot filter out non-terminating computation since termination is non-decidable
- In open systems, users cannot be assumed to simply well-behave

→ Need to prevent/discourage users from deploying/invoking infinite or long computations
Ethereum and Gas

Ethereum proposes gas, i.e., making users pay for computation executions:

- TXs are endowed with two more fields: gasLimit & gasPrice
  - Miners could delay TXs having a low gasPrice
  - Users can increase their priority by increasing gasPrice

- Upon execution, each bytecode instruction increases the $g$ counter
  - according to a price list defined in [28]

- Whenever $g > \text{gasLimit}$ an exception is raised, reverting side effects

- In any case, upon termination, the issuer balance is decreased of $\Delta\text{ETH} = \text{gasPrice} \cdot g$
  - The winning miner can redeem $\Delta\text{ETH}$ as a compensation for its computational effort

The economical dimension of computation has to be taken into account when designing Ethereum smart contracts
Ethereum smart contract example with Solidity

```solidity
contract Counter {

  event Increased(uint oldValue, address cause);

  address owner; uint value;

  function Counter() public { owner = msg.sender; } // <-- constructor

  function inc(uint times) public { // <-- API
    for (uint i = 0; i < times; i++) {
      emit Increased(value++, msg.sender);
    }
  }

  function kill() public { // <-- API
    require(msg.sender == owner);
    suicide(owner);
  }

  function () { // <-- fallback
    throw;
  }
}
```

4 https://solidity.readthedocs.io
Ethereum smart contract example with Solidity

State 0
- Alice\textsubscript{balance} = 20
- Bob\textsubscript{balance} = 20

State 1
- Alice\textsubscript{balance} = 20
- Bob\textsubscript{balance} = 17
- Carl\textsubscript{balance} = 5
- Counter\textsubscript{owner} = Alice
- Counter\textsubscript{value} = 0
- Dan\textsubscript{balance} = 3

State 2
- Alice\textsubscript{balance} = 20
- Bob\textsubscript{balance} = 16.89
- Carl\textsubscript{balance} = 5
- Counter\textsubscript{value} = 3
- Dan\textsubscript{balance} = 2.83
- Eve\textsubscript{balance} = 5.28

State 3
- Alice\textsubscript{balance} = 20
- Bob\textsubscript{balance} = 16.89
- Carl\textsubscript{balance} = 4.83
- Counter\textsubscript{value} = 5
- Dan\textsubscript{balance} = 8.07
- Eve\textsubscript{balance} = 4.93
Smart contracts issues I

No privacy or secrets

Every information ever published on the blockchain stays on the blockchain

- The private state of a smart contract is not secret
- Pseudo-anonymity can be broken with statistics & data-fusion
- Illegal/anti-ethic behaviour can be revealed years later

! No secret voting?!
Poor randomness

It is difficult to achieve (pseudo-)randomness because of the lack of trustable sources

- Real randomness cannot be employed (replicas would diverge)
- Most of the blocks observable information are under the control of the miner
  - e.g. timestamp, height, hash, etc.
- The block hash seems a good choice
  - but this is an egg-and-chicken problem

! No lottery?!
Smart contract inter-communication

Can a SC interact with another one? Which is the exact semantics of doing so? Is OOP the best programming paradigm?

In Ethereum, SC are essentially objects communicating by means of synchronous method calls. The callee SC are referenced by callers by means of their address:

- the control flow originating from a user may traverse more than a SC
- the caller waits for the callee
- unattended re-entrancy if difficult to avoid \[4, 20\]
- and it may lead to undesired behavioural subtleties and frauds \[13\]

https://medium.com/gus_tavo_guim/reentrancy-attack-on-smart-contracts
Smart contracts issues IV

Impossibility to fix bugs

SC code is immutable. Immutability is both a blessing and a curse. Buggy contracts cannot be fixed, updated, replaced, or un-deployed

- Buggy, misbehaving, fraudulent SCs will remain so, wasting miners resources
- Paramount importance of correct-design and formal validation
  - a problem *per se* in Turing-complete languages
- Behavioural OOP design patterns are possible, but critical because of the previous issue
Lack of proactiveness

SCs are purely reactive computational entities

- They always need to borrow some user’s control flow
- They are time-aware but not reactive to time
- They cannot schedule or postpone computations
  - no periodic computation (e.g. payment)
Disembodiement [21] & lack of concurrency

Computation is logically located everywhere and transactions are strictly sequential. This may be a wasteful approach in some cases:

- Independent computation cannot be executed concurrently
- Computations only making sense locally need to be replicated globally
- Heavy computations cannot be splitted into parts to be run concurrently
Granularity of computation-related costs

Ethereum is not the first platform applying a price to computation:

* e.g. Common practice on the Cloud, and the X-as-a-Service paradigm

BTW, is the instruction-level cost granularity the better one?

* e.g. In the most trivial implementation of a publish-subscribe mechanism, it is the publisher paying the variable price
Programming paradigms for smart contracts

Problem

SCs research care a lot about HLLs but some issues are related to their underlying operational semantics:

- Synchronous calls are usually hard coded by construction
- Poor care for what concerns inter-SC interaction
- Lack of control flow encapsulation
- Lack of proactiveness

Goals

Investigating the adoption of interaction-friendly paradigms such as actors or agents
Smart contracts as Actors I

Possible modification to SCs operational semantics:

- Asynchronous message passing as the unique means of inter-SC communication + control flow encapsulation
  - The total ordering of events perfectly matches the event-loop semantics of Actors

- Sending a message implies issuing an invocation TX
  - Analogously to what users do
  - Messages are sent only after the current TX terminates correctly

- Selective/guarded receive for enabling or delaying some computation

- Private, synchronous call are still possible
  - Can be used to implement pure computations
Interesting questions arising from this vision:

? Who is paying for SC-initiated control flows?
? How to compensate miners for delayed computations?

Possible activities

- Re-thinking or editing some BCT formal semantics in terms of actors
- Forking some existing BCT project to inject the actors semantics
- Designing (and developing?) a novel BCT project exposing an actor-based SC abstraction
There seems to be more degrees of freedom here:

- Different, possibly overlapping, declination of the **Agent** notion:
  e.g. Believes-Desires-Intentions (BDI), Agents & Artifacts (A&A)

- Different possible mappings are for: \( \{ \begin{align*}
  \text{Agent} \\
  \text{Environment} \\
  \text{Artifact?}
\end{align*} \)

- Which choices are the best ones and why?
For instance, let’s image SCs as BDI agents:

- Then, what’s the environment? What can an agent perceive?
- Is goal-oriented reasoning useful in this context?
  - Should a SC reason about how to execute its business logic?
- What about epistemic actions?
  - Should a SC ask for unknown informations to other (human?) agents?
- Do multiple intentions (i.e., multiple control flows) make sense?
  - Who is paying for them?
  - Who is in charge for executing them? Using which concurrency model?

Possible activities

- Re-thinking or editing some BCT formal semantics in terms of agents, environment, artifacts, etc.
Logic-based smart contracts I

Problem

SCs currently lack:

- high level understandability in their HLLs
- observability of the deployed source code
- some degree of evolvability enabling them to be modified (or fixed)

Goals

Investigating how the adoption of a logic interpreted language (e.g. Prolog) may improve SC for what concerns such aspects
Logic-based smart contracts II

Employing a logic language, such as Prolog, introduces some benefits:

- naturally **declarative** and **goal-oriented**, improving understandability
- **static KB** for the immutable code, **dynamic KB** for the mutable part
- **asserts** & **retracts** only affect the dynamic KB
  - thus enabling some sort of **controlled** evolvability
- being an interpreted language it always possible to inspect the KB
  - without disassemblers
- **guarded/selective receive** to enforce a boundary for SCs API
- **context-aware predicates** for inspecting the current context
  - similarly to Solidity’s Globally Available Variables
Logic-based smart contracts III

... And some more questions:

- should the computational economic cost model be re-designed to embrace LP basic mechanisms?

- how should logic SCs interact?

Possible activities

- Re-thinking or editing some BCT formal semantics to embrace such a vision
- Designing (and develop) such a novel vision from scratch
Blackboard-based approaches and smart contracts

Opportunity

Shared blackboards systems may take real advantage of the replication and fault-tolerance features they would inherit if deployed on top of a BCT layer. For instance:

- e.g. tuple-based coordination
- e.g. distributed logic programming

Goals

- Investigating whether BCTs are useful in such contexts or not.
- Considering such contexts as applications, looking for improvements to the BCTs
Can we build the archetypal LINDA model on top of BCTs?

- If yes, tuple spaces would inherit a lot of desirable properties
  - e.g. Decentralisation & replication, fault-tolerance, consistency, etc.

- Which computational economical cost model for LINDA primitives?
- How to handle control flow-related aspects?
  - e.g. suspensive semantics
- Can we inject programmability too?

Figure: Our vision: BCTs as the backbone on top of which communication and coordination services are built
Tuple-based coordination on the Blockchain II

Possible activities

- Compare several BCTs from the coordination-capabilities point of view, modelling and implementing LINDA on top of them
- Compare several BCTs from the coordination-capabilities point of view, modelling and implementing ReSpecT on top of them
Distributed LP on the Blockchain

Can we employ the blockchain as a blackboard enabling distributed agents to cooperatively participate to some SLD reasoning process?

- Again, desirable properties would be “automatically” inherited
- LP-friendly economical incentives/disincentives could be conceived stimulating miners to adopt a particular strategy when building/exploring some proof-tree
- Concurrent LP has some well-known critic aspects
e.g. AND-parallelism, OR-parallelism, termination, non-termination, shared variables

? How to handle KB mutability while reasoning?

Possible activities

- Develop (at least) a proof of concept or sketched implementation showing the feasibility of concurrent, blockchain-based, SLD resolution process
Formal (meta-)model for BCTs and smart contracts

Problem
A part from Ethereum, other mainstream BCTs lack a formal semantics specification. Furthermore, a general meta-model comprehending them all is still missing.

Goals
- Defining a meta-model explaining all (or most) existing BCTs or proving it to be impossible
- Defining an operational semantics for all (or most) existing BCTs
- Showing why the operational semantics of each BCT is an instance of the general meta-model

Possible activities
- SLR about the formal semantics of one or more BCTs
- Define your own formal semantics/meta-model
Simulating the blockchain

Problem
Some local consensus approaches lack formal theorems proving their properties or their sensibility to the parameters variation e.g. $\Delta T$, $CP$ distribution, economical cost model, etc.

Goals
Designing & developing an agent-based simulation framework where such interrelated aspects can be studied \textit{in silico}

Possible activities
- Develop the simulation framework and show its effectiveness by simulating a simple consensus model
- Design a complex consensus model to be simulated on the aforementioned framework to reveal critical parameters regions
Local consensus mechanisms

Problem
Classical BFT consensus algorithms are very powerful but their performance essentially degrades with the amount of nodes.

Goals
Conceive, design, implement, and assess other local (stochastic?) consensus mechanisms ensuring some (possibly provable) security properties.

Possible activities
- SLR on classical/novel consensus mechanisms: compare & classify
- Implement some classical/novel consensus protocol
- Design your own (non-trivial) consensus mechanism
Concurrent execution of independent SCs
- data and computation partitioning on different nodes
- branching/merging of the blockchain (making it a DAG)

Possible activities
- SLR on such aspects
- Design your own (non-trivial) concurrent BCT
Privacy & confidentiality for smart contracts

Problem
SCs lack confidentiality when interacting with users, and some means to hide their private internal state

Goals
Developing a cryptographic schema aimed at injecting some degree of confidentiality/privacy into smart contracts

Possible activities
- SLR on privacy/confidentiality-related aspects
- Design your own (non-trivial) cryptographic schema


A survey of attacks on Ethereum smart contracts (SoK).
The Next 700 BFT Protocols.

Hashcash - A Denial of Service Counter-Measure.

Towards robust distributed systems (abstract).

Blockchain Consensus Protocols in the Wild.

Practical byzantine fault tolerance and proactive recovery.
References


References


References VI


A gentle introduction to the Blockchain and Smart contracts

Giovanni Ciatto  
{ giovanni.ciatto@unibo.it }

Talk @ Autonomous Systems Course, A.Y. 17/18
Dipartimento di Informatica, Scienza e Ingegneria—Università di Bologna

May 30, 2018