Blockchain & Smart Contracts
Basics and Perspectives for MAS

Andrea Omicini
andrea.omicini@unibo.it

Giovanni Ciatto
giovanni.ciatto@unibo.it

Doctorate Mini-School @ WOA 2018

Palermo, Italy – 27 June 2018

Next in Line...

Focus on...

State Machine Replication (SMR) I

Main idea [Schneider, 1990, Charron-Bost et al., 2010]

- executing the same (not necessarily finite) state machine
- ! where state machine $\approx$ program
- over multiple independent (possibly distributed) processors
- concurrently & consistently
- in order to achieve
  - fault tolerance—to stops, crashes, lies, bugs, etc,
  - availability and reactivity
  - data / software replication & untamperability

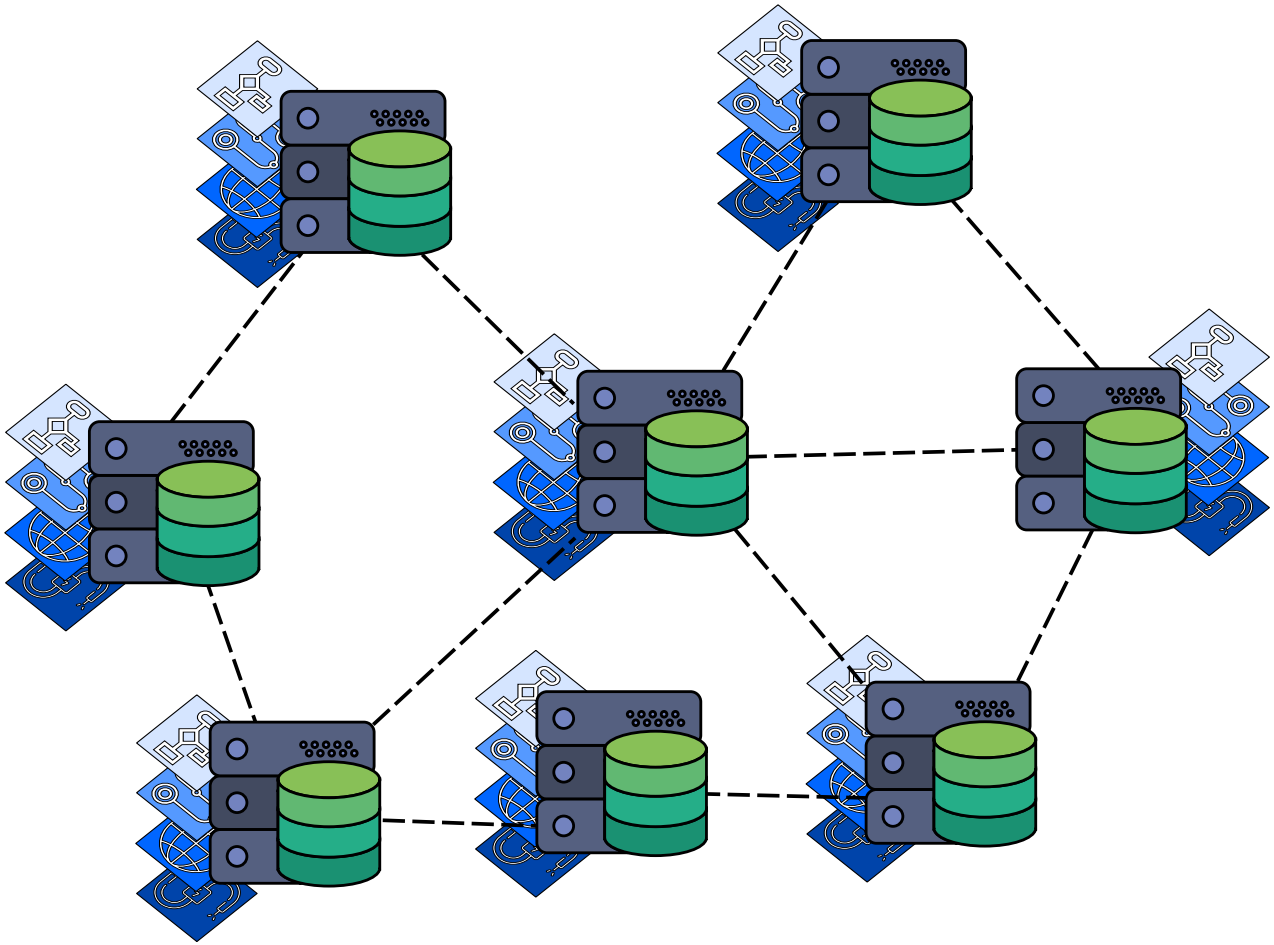
State Machine Replication (SMR) II

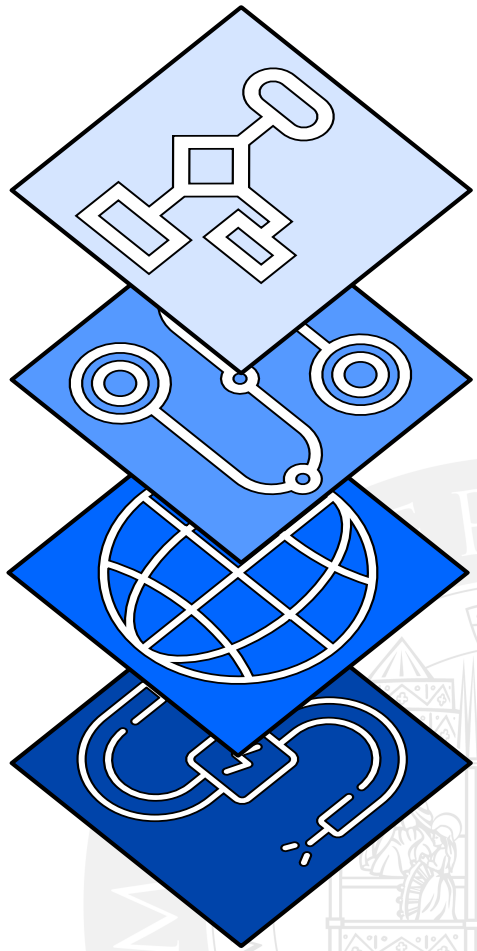
A network of replicas

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) III





Deterministic State*less* Computations

$Input \longrightarrow Computation \longrightarrow Output$

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

- can be arbitrarily replicated
- replicas can be run concurrently

Deterministic

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

Non-deterministic

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

Deterministic State*ful* Computations I

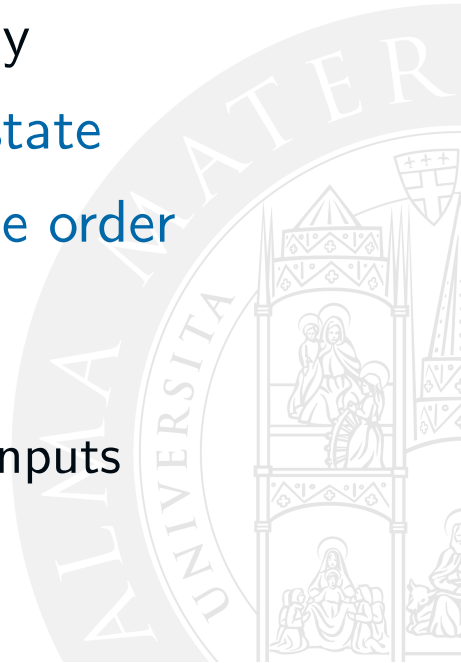
$$(Input, State) \longrightarrow Computation \longrightarrow (Output, State')$$

Computation is deterministic if it *always* produces the same output when it is performed on the same input and state.

- can be arbitrarily replicated & replicas run concurrently
- all replicas must be initialised within the same initial state
- all inputs must be submitted to all replicas in the same order

! input $\approx$ method call

  - they all move through the same sequence of states
  - maintaining state consistency of the state among on inputs



Deterministic State*ful* Computations II

Deterministic

```
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;
  }
}
```

Deterministic State*ful* Computations III

Non-deterministic

```
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;
  }
}
```

Deterministic State*ful* Computations IV

The blockchain...

... is essentially a means for
replicating deterministic stateful
computations

"Universal" State Machine Replication I

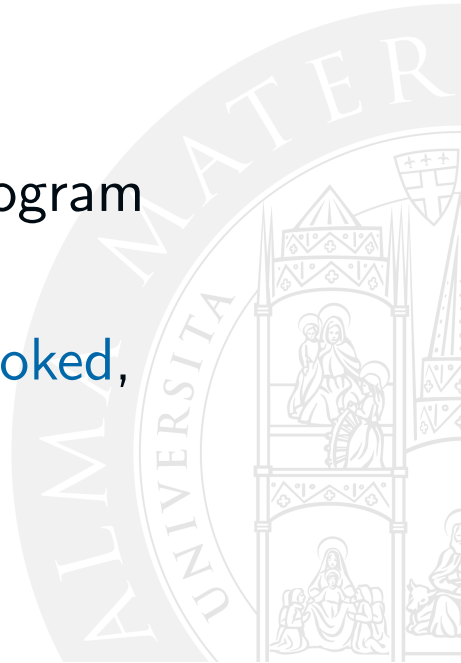
$$UTM : TM = Interpreter : Program = ? : SMR$$

! UTM $\stackrel{def}{=}$ Universal Turing Machine — TM $\stackrel{def}{=}$ Turing Machine

- we can replicate a stateful deterministic program implementing a particular business logic

  e.g., a bank ledger

- in the same way, we could replicate a deterministic program implementing an interpreter
- whose API would enable programs to be deployed, invoked, undeployed, etc.



"Universal" State Machine Replication II

Smart contracts

- smart-contracts-enabled blockchains essentially act as replicated "universal" state machines on which smart contracts (SC) can be deployed
  - where smart contract $\approx$ a program deployed on such a machine

"Universal" State Machine Replication III

Example

```
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;
  }
}
```

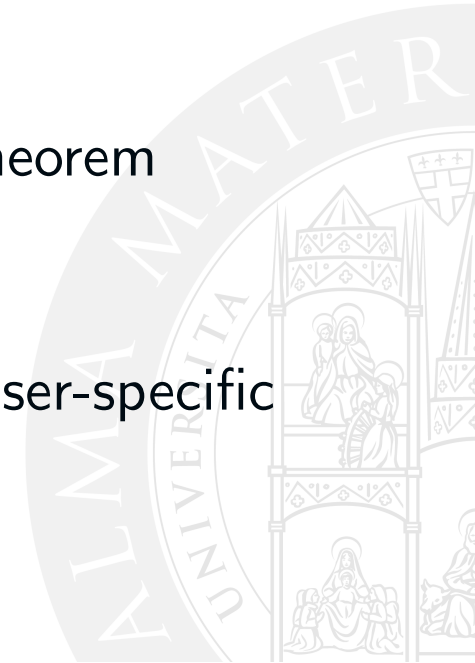
SMR & *Distributed* Systems

- in a distributed system, messages may be lost, reordered, or duplicated by the network
  - each node may perceive a different view about the system events
- lack of global time

  $\Longrightarrow$ lack of total ordering of events

  $\Longrightarrow$ lack of a trivial notion of consistency
- *consistency*, *availability*, *partition*-resistance (CAP) theorem [Brewer, 2000]

  $\Longrightarrow$ you cannot have more than two of them
- *authentication* is required if the replicated service is user-specific



SMR, Middleware & Consensus I

Middleware and consensus

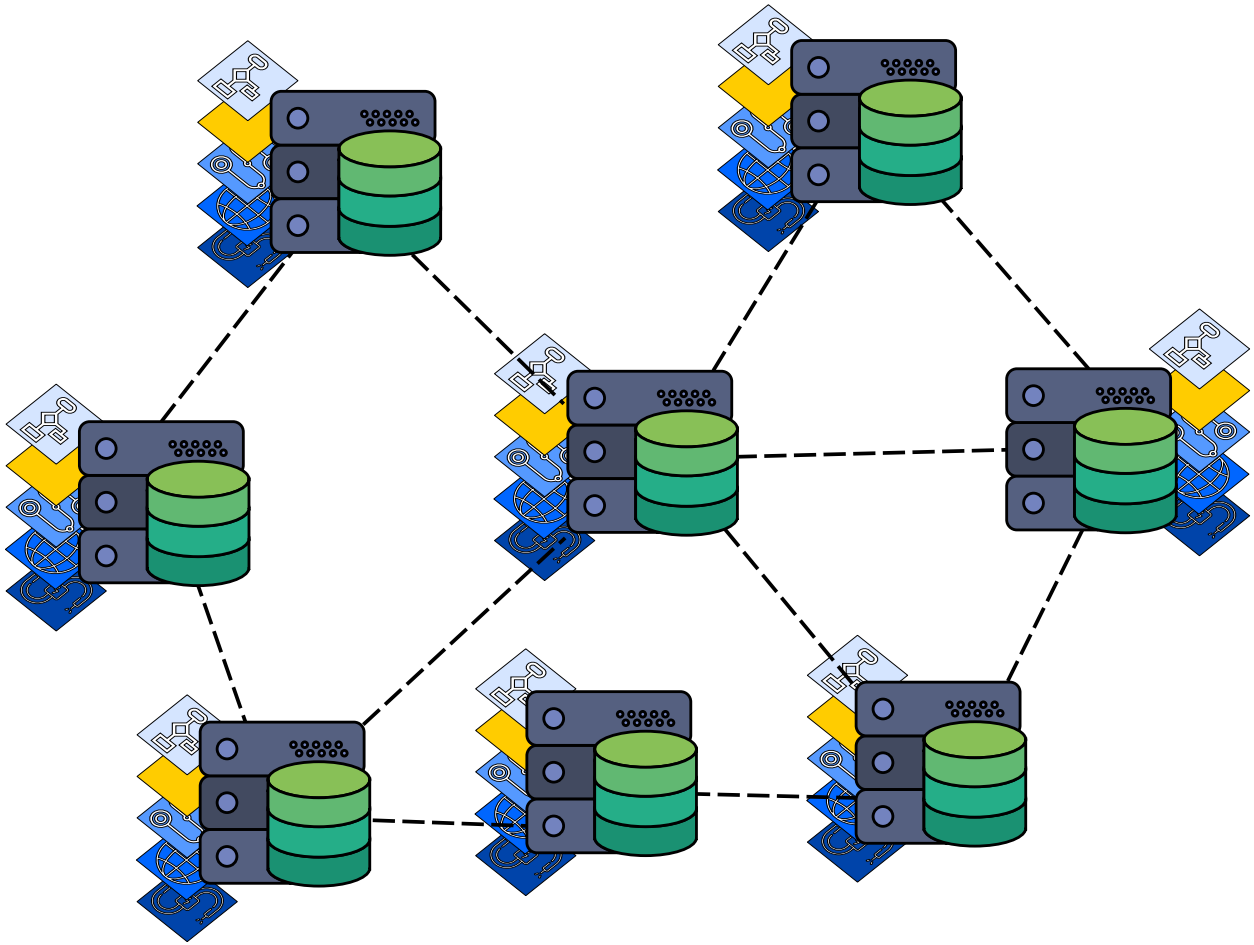
Each replica is executed on top of a middleware taking care of validating & ordering inputs for the replicated program

- 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

! Fischer, Lynch and Patterson (FLP) theorem [Fischer et al., 1985]

$\Longrightarrow$ impossibility of consensus without timing assumptions

SMR, Middleware & Consensus II



SMR & *Open* Distributed Systems I

World-wide openness & trust issues

- 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.
- yet, we can tolerate *some* byzantine nodes
  - ! *less than 1/3 of the total amount of nodes*, according to Lamport's Byzantine Generals Problem solution [Lamport et al., 1982]
- we can also ease the recognition of prohibited or unauthorised behaviours by employing cryptography
  - e.g. pub/priv key pairs for user authentication
  - e.g. 1-way hash functions and MAC for data integrity

SMR & *Open* Distributed Systems II

Takeaway

- the blockchain is a smart way to achieve (U)SMR
- dealing with – i.e., *mitigating* – well-known issues of open distributed systems

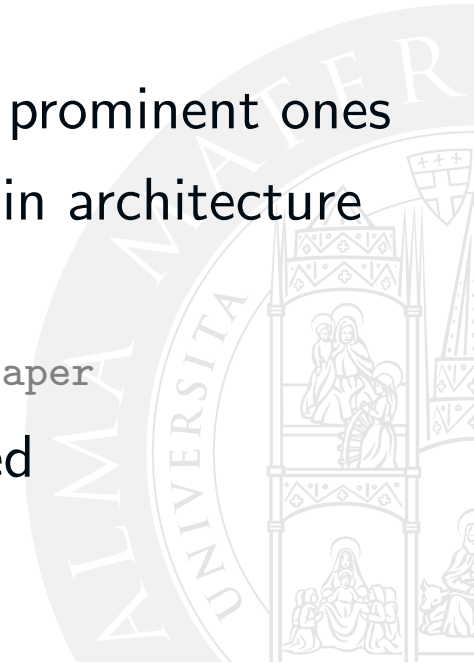
Focus on...

Disclaimer

- most of blockchain-related works focus on a *specific* blockchain technology (BCT henceforth) using a bottom-up approach.
- this approach hinders generality and limits the discussion about what we can do on top of BCT
- in the following we present the blockchain in a top-down way, gathering from a number of sources—being [Nakamoto, 2008, Wood, 2014, Androulaki et al., 2018] the most prominent ones
- furthermore, the following description of the blockchain architecture and workings is strongly inspired to Ethereum

  `http://github.com/ethereum/wiki/wiki/White-Paper`

  —probably the most mature, studied, and documented smart-contracts-enabled BCT



BCT

Blockchain technology (BCT)

An 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

An 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 I

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 certification authority (CA)
- *identifiers* are also known as addresses in this context

Permission*ed vs.* permission*less*

- either each user owns multiple non-intelligible identifiers. . .
  - ✓ pseudonymity
  - ✓ decentralised
  - × Sybil-attack [Douceur, 2002]
- . . . or he/she owns a single certified identifier
  - × single point of failure/trust

Entity Identifiers II

! *smart-contracts-enabled BCTs identify both smart contracts' instances and users by means of the same sort of addresses*

The World State I

System state

$$\begin{array}{rcl} \langle \mathit{SystemState} \rangle & ::= & \texttt{entityID} \mapsto \langle \mathit{Account} \rangle \\ & | & \langle \mathit{SystemState} \rangle \ \langle \mathit{SystemState} \rangle \end{array}$$

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

The World State II

Account state

e.g. $\langle Account \rangle ::= (\texttt{balance}, \langle Storage \rangle, \langle Code \rangle, \langle Metadata \rangle)$

- the account state *conceptually* consists of several fields keeping track of what a particular entity currently owns
- fields may vary depending on
  - the nature of the blockchain
  - whether the entity is a smart contract or a user

e.g., BCTs coming with native cryptocurrencies, usually keep track of account `balance`s (at least)

e.g., smart-contracts-enabled BCTs may keep track of their source code and internal $\langle Storage \rangle$

The World State III

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

  void deposit(String userID, int money) {
    //                ^^^^^^
    //          entity identifier

    balances[userID] += ...
    //^^^^^^^^^^^^^^
    //account state
  }

}
```

The World State IV

Many possible implementations

- Unspent Transaction Output (UTXO)

  e.g., Bitcoin [Nakamoto, 2008]
- account-based

  e.g., Ethereum [Wood, 2014]
- key-value store

  e.g., Hyperledger fabric [Androulaki et al., 2018]

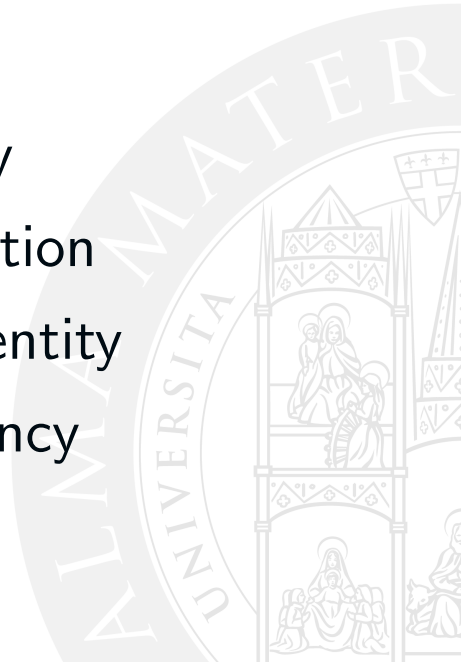
Transactions (a.k.a. Inputs or Messages) I

Informal definition

$$\langle \textit{Transaction} \rangle ::= (\texttt{txID}, \texttt{issuerID}, \langle \textit{Signature} \rangle, \texttt{recipientID}, \texttt{value}, \langle \textit{Data} \rangle)$$

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

- `txID` — the transaction progressive number
- `issuerID` — the address of the transaction issuer entity
- $\langle \textit{Signature} \rangle$ — the cryptographic signature of the transaction
- `recipientID` — the address of the transaction recipient entity
- `value` — some non negative amount of cryptocurrency
- $\langle \textit{Data} \rangle$ — some arbitrary data



Transactions (a.k.a. Inputs or Messages) II

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

deployment TX — if $\texttt{recipientID} = \varnothing \wedge \langle \mathit{Data} \rangle = \texttt{code}$

invocation TX — if $\texttt{recipientID} = \texttt{address} \wedge \langle \mathit{data} \rangle = \texttt{whatever}$

money transfer TX — if $\texttt{recipientID} \neq \varnothing \wedge \texttt{value} > 0 \wedge \langle \mathit{Data} \rangle = \varepsilon$

Transactions life-cycle (Part I)

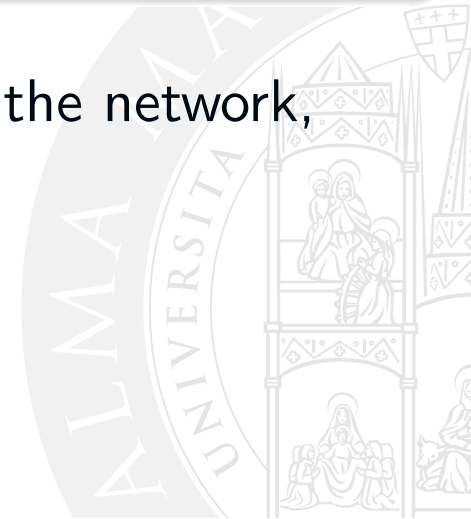
1. a issuer user compiles a transaction
2. he/she signs it with his/her private key K_{pr}
3. his/her node spreads the transaction over the P2P network
4. peers only take into account valid transactions
5. ...

Transactions (a.k.a. Inputs or Messages) III

Transactions validity

- 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 & Block Chains I

Informal definition

$$
\begin{array}{rcl}
\langle \mathit{Block} \rangle & ::= & (\texttt{prevHash}, \texttt{index}, \texttt{time}, \langle \mathit{TxList} \rangle, \langle \mathit{FinalState} \rangle) \\
\langle \mathit{TxList} \rangle & ::= & (\langle \mathit{Transaction} \rangle, \langle \mathit{IntermediateState} \rangle) \\
& | & \langle \mathit{TxList} \rangle \langle \mathit{TxList} \rangle
\end{array}
$$

Blocks are timestamped, hash-linked lists of transactions.

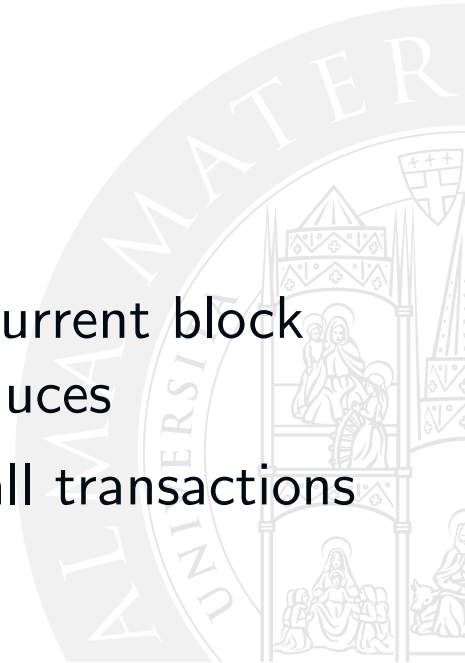
`prevHash` — the hash of the previous block

`index` — the index of the current block

`time` — the timestamp of the current block

$\langle \mathit{TxList} \rangle$ — the list of transactions included into the current block and the intermediate system states they produces

$\langle \mathit{FinalState} \rangle$ — the system state resulting from applying all transactions in $\langle \mathit{TxList} \rangle$, respecting their order



Blocks & Block Chains II

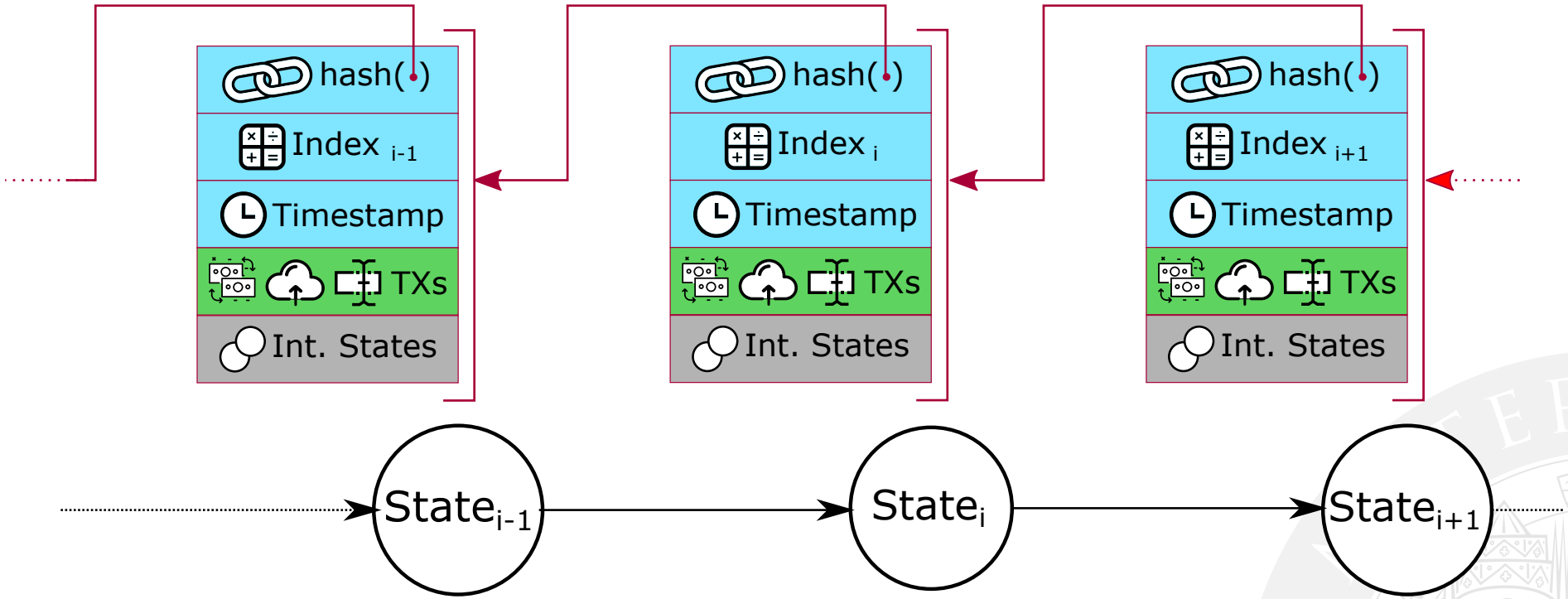


Figure: Graphical representation of the Block-chain from the global p.o.v.

Blocks Features

- replication + hash-chaining $\rightsquigarrow$ untamperability of the past
- hash-chain + time + ordering $\rightsquigarrow$ timestamping / notary service
  [Haber and Stornetta, 1991]
- hash-chain + cryptographic signatures $\rightsquigarrow$ $\begin{cases} \text{accountability} \\ \text{non-repudiation} \end{cases}$
- they are supposed to be published (almost) periodically

! in the general case $\lim_{n\to\infty} \mathbb{P}[\mathit{inconsistent}(B_i)] = 0$, where
  - B_i is the i^{th} block
  - n is the amount of successor blocks of B_i
  - $\mathit{inconsistent}(B_i)$ 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 I)

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 II)

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)

! *Those life-cycles may vary a lot depending on the specific consensus algorithm employed*

The Network Point of View

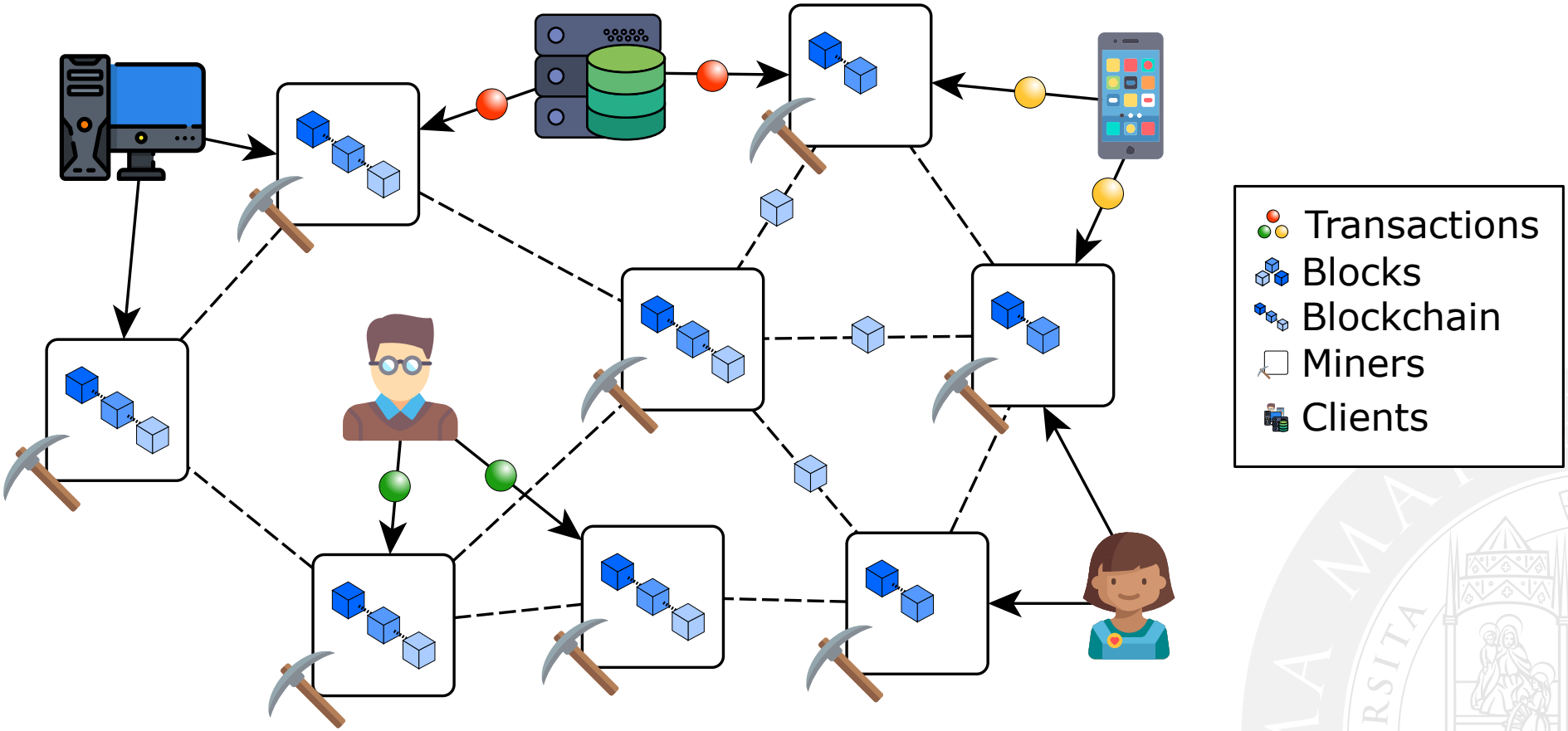


Figure: Graphical representation of the Block-chain from the network p.o.v.

Consensus & Mining I

Permission-*ed* BCTs

constrain users' IDs through CAs

⇓

"classical" quorum/leader-based consensus algorithms

- BFT algorithms
  - e.g., PBFT [Castro and Liskov, 2002]
  - e.g., BFT-SMaRt [Sousa and Bessani, 2012]
  - e.g., HoneyBadger BFT [Miller et al., 2016]
- non-BFT algorithms
  - e.g., Paxos [Lamport, 1998]
  - e.g., Raft [Ongaro and Ousterhout, 2014]
  - 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 [Back, 2002]
- e.g., Proof-of-Stake [Proof-of-Stake, 2017]
- e.g., Proof-of-Elapsed-Time [Chen et al., 2017]
- e.g., IOTA Tangle [Popov, 2017]

Comparisons & surveys in

[Aublin et al., 2015],

[Cachin and Vukolić, 2017]

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 multiadministrative organisations

Permission-*less* BCTs

"novel" competition-based approaches

- no assumption on N

  UB virtually ∞

- low throughput

  OoM $\sim$ 10 TXs/s

- probabilistic consistency
- ideal for open systems

Proof-of-Work (PoW) I

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 ΔT seconds
  - ₿ finding a block hash having a given amount of leading zeros
  - Ξ hashing (pseudo)random pieced of data attained form 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
  - Ξ Greedy Heaviest Observed SubTree (GHOST [Sompolinsky and Zohar, 2013])

Proof-of-Work (PoW) II

PoW relevant features

- competition-based, local, eventually-consistent, stochastic approach
- self-adaptive mining difficulty, s.t. $\mathbb{E}[\Delta T] = const$
  - ! the system update frequency is $1/\mathbb{E}[\Delta T]$
- only computing power (CP) matters here
  - Sybil-attack resistant
  - CP distribution & majority rule (51% attack) [Eyal and Sirer, 2014]

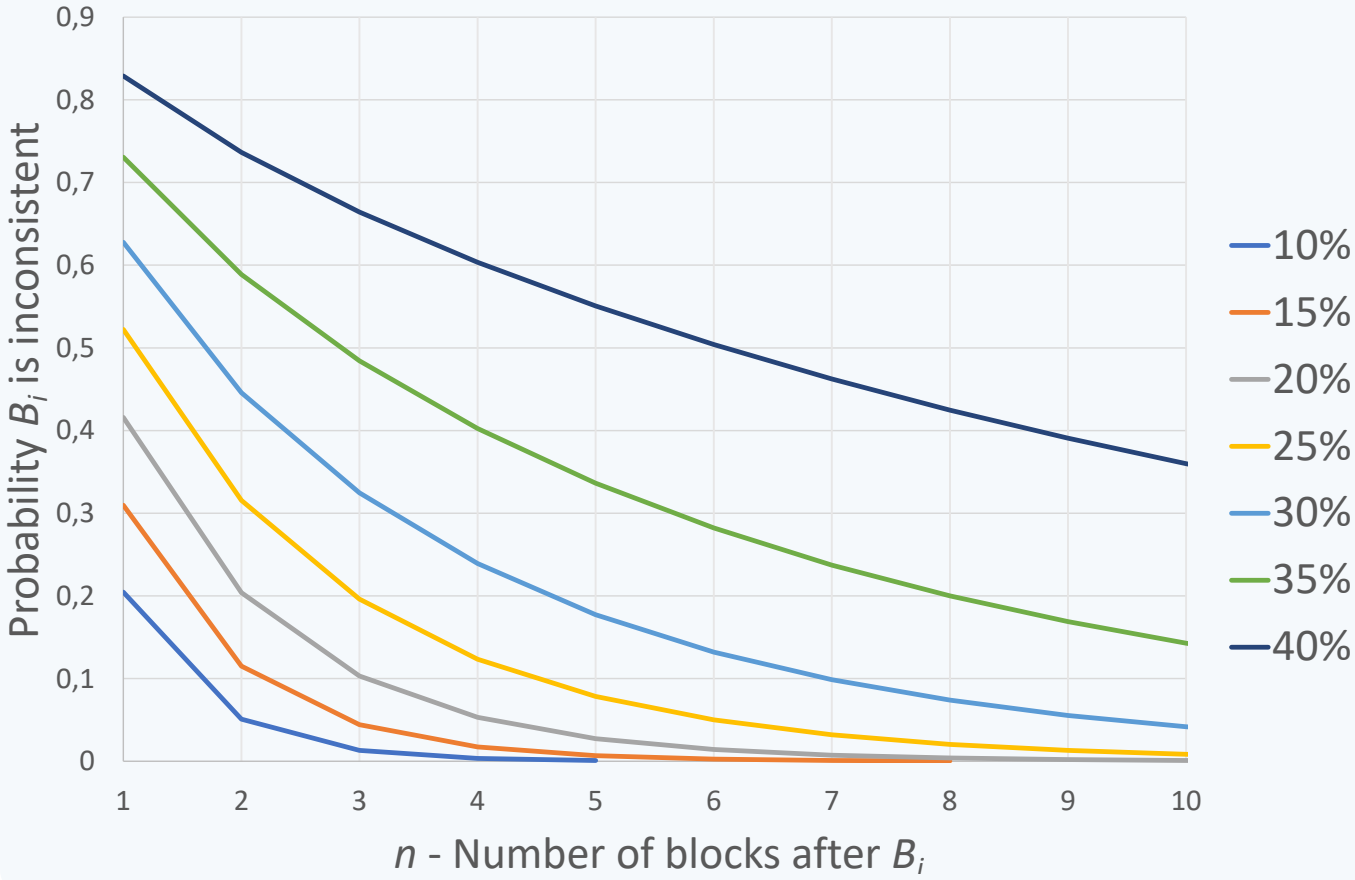
! endows the cryptocurrency with its economical value

! miners require economical compensation for their effort

Proof-of-Work (PoW) III

PoW security

$r = \frac{adversary_{CP}}{honests_{CP}}$ $\mathbb{P}[n \mid r] = 1 - \sum_{k=0}^{n} \frac{(nr)^k e^{-nr}}{k!}(1 - r^{n-k})$ [Nakamoto, 2008]



in Bitcoin

- $n_{threshold} = 6$
- $\approx 1h$ since $\mathbb{E}[\Delta T] = 10m$
- 99.999% secure if $adversary_{CP} <$ 13% $total_{CP}$

Focus on...

Smart Contracts [Szabo, 1996]

Informal definition

A smart contract (SC) is a *stateful*, *reactive*, *user-defined*, *immutable*, and deterministic processes executing some *arbitrary* computation *on the blockchain*, i.e., while being replicated over the blockchain network

stateful — SC encapsulates its own state–like an object in OOP

reactive — SC can only be triggered by issuing some invocation TX

user-defined — users can deploy their SCs implementing an arbitrary logic by issuing a deployment TX

immutable — SC source/byte-code cannot be altered after deployment

arbitrary — SC is expressed with a Turing-complete language

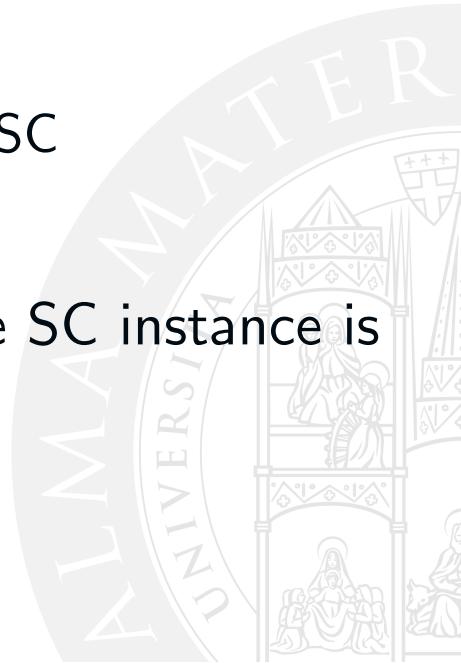
replicated — the blockchain is essentially a replicated interpreter, employing a consensus protocol to synchronise SC replicas

SC Features & Expectations

- immutability + untamperability + accountability + decentralisation
  $\Longrightarrow$ SC can be trusted in handling financial operations between organisations
  - even easier with native cryptocurrency
- the code is always right, the true history is on the blockchain
  - reducing disputes
  - removing the need for arbitration
- lack of situatedness: totally disembodied [Mariani and Omicini, 2013] data & computation
  - 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 organisations (DAOs), etc.

SC Deployment

1. initially there is no smart contract
   - i.e., the system state includes no SC entity
2. a user can instantiate a SC 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



SC Invocation

1. users can trigger already-deployed SCs by publishing invocation TXs
   - specifying the SC address as recipient
   - providing some input data specifying the requested computation
2. eventually, each node will receive the TX and execute it
   - the SC code usually decides what to do given the provided input data
3. 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
4. the wrapping block is eventually confirmed by the consensus protocol, and the invoked computation can be actually considered executed

Issues from Turing-completeness

What if the program below is a SC?

```
class Bomb {
  int i = 0;

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

- BCTs cannot filter out non-terminating computation since the halting problem is undecidable over Turing machines
- in open systems, users cannot be simply assumed to well-behave

$\Longrightarrow$ need to prevent/discourage users from deploying/invoking infinite or long computations

Termination by Resource Consumption

Ethereum & Gas

Ethereum users pay gas for executing computations

- 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 [Wood, 2014]
- whenever $g >$ `gasLimit` an exception is raised, reverting side effects
- in any case, upon termination, the issuer balance is decreased of $\Delta ETH =$ `gasPrice` $\cdot\, g$
  - winning miner can redeem ΔETH as a compensation for its computational effort

! *The economical dimension of computation has to be taken into account when designing Ethereum smart contracts*

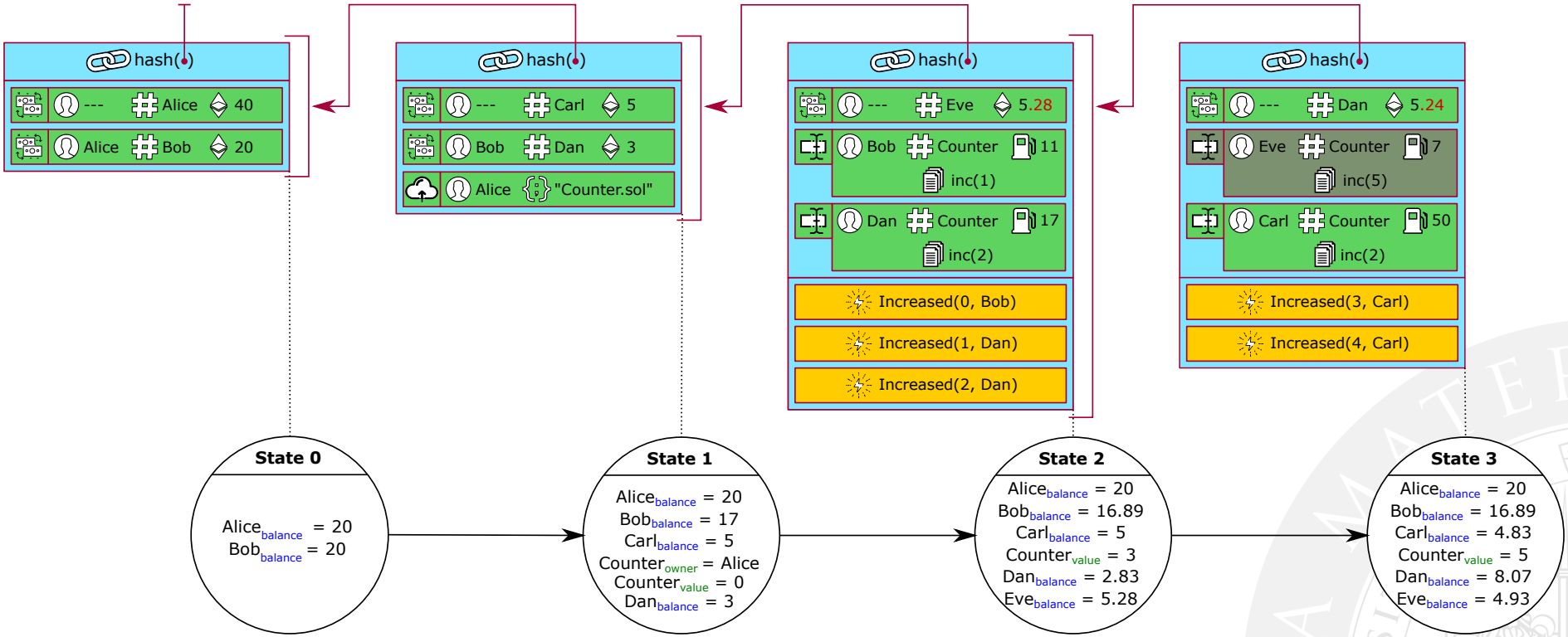
SC Example: Ethereum with Solidity I

An Ethereum SC in Solidity

http://solidity.readthedocs.io

```
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;
  }

}
```

SC Example: Ethereum with Solidity II



SC Issues I

Neither privacy nor secrets

Every information ever published on the blockchain stays on the blockchain forever

- 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?!?

SC Issues II

SC inter-communication

- some questions:
  - can a SC interact with another one?
  - in case, which semantics?
  - is OOP the fittest programming paradigm?
- in Ethereum, SC are essentially objects communicating by means of synchronous method calls
- the callee SC is referenced by callers by means of its address
  - control flow originating from a user may traverse more than a SC
  - the caller waits for the callee
  - unattended re-entrancy difficult to avoid [Atzei et al., 2017, Luu et al., 2016]
  - may lead to undesired behavioural subtleties and frauds [Dupont, 2017]

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

SC Issues III

No way 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, fraudolent SCs will remain so, wasting miners resources

→ correct-design and formal validation is of paramount importance

- a problem *per se* in Turing-complete languages

- behavioural OOP design patterns are possible, yet critical because of the previous issue

SC Issues IV

Lack of proactiveness

SCs are purely reactive computational entities

- they always need to borrow some end 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 [Mariani and Omicini, 2013] & 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 split into parts to be run concurrently

SC Issues V

Poor randomness

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

- real randomness cannot be employed (replicas would diverge)
- most of the block observable information are under the control of the winning miner miner

  e.g. timestamp, index, etc.
- the block hash is apparently a good choice
  - but this is an egg-and-chicken problem

! no lottery?!?

SC Issues VI

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
- is the instruction-level cost granularity the best one?
- what about pluggable, business-oriented cost models?
  e.g., in the most trivial implementation of a publish-subscribe mechanism, it is the publisher that pays the variable price

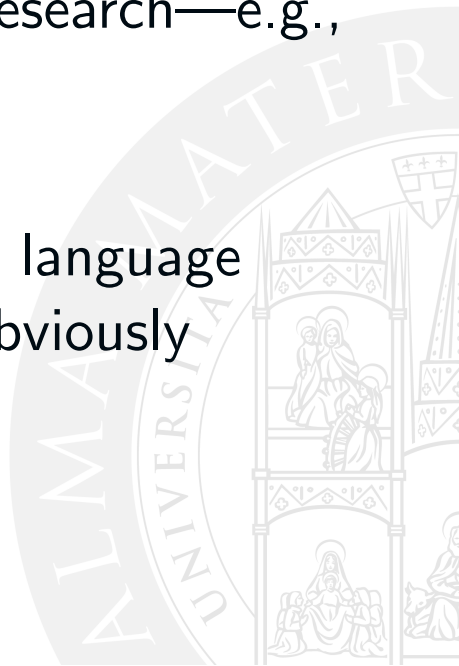
Next in Line...

Conceptual Premises

- notions such as SMR and USMR, as well as technologies such as BCT and SC are currently exploited mostly for financial transactions, identity management, or asset property tracking
- however, the blockchain can be generally seen as a novel approach to distributed systems, enabling a common, consistent view of some shared state among distributed agents
- also, SCs can be seen as an emerging computational model allowing users to deploy arbitrary computations on a blockchain-based system
- more general applications for BCTs and SCs are definitely possible
- and, they have the potential to open lines of some interest for many of the researchers who constitute the WOA community nowadays

Programming Languages for Smart Contracts I

- the language used to write SC affects the potential features of SC—e.g, [Idelberger et al., 2016]
- the language for programming SC is a typical line of research—e.g., [Seijas et al., 2017]
- issues range from underlying mechanisms to high-level language abstractions—including the programming paradigm, obviously



Languages Mechanisms for Smart Contracts

- some language issues are related to their underlying operational mechanisms
- for instance
  - synchronous calls are usually *hard coded* by construction
  - poor care for what concerns inter-SC interaction
  - lack of control flow encapsulation
  - lack of proactiveness

Possible lines of development

- investigating the adoption of interaction-friendly paradigms such as actors or agents
- where some (all) of the above limitations are overcome at the fundamental programming abstraction level

Smart Contracts as Actors I

Changing the SC operational semantics

- asynchronous message passing as the unique means of inter-SC communication + control flow encapsulation
  - total ordering of events 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?

Smart Contracts as Actors II

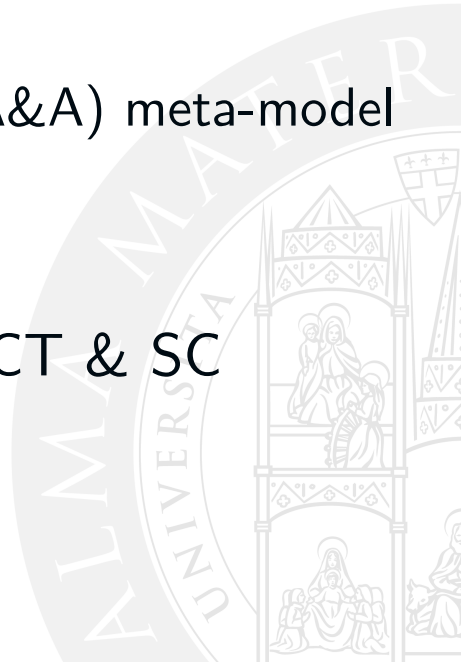
Potential research & technical activities

- re-thinking or editing some BCT formal semantics in terms of actors
- forking some existing BCT project to inject the actors semantics
- designing (developing) a novel BCT project exposing an actor-based SC abstraction

Smart Contracts & MAS I

Multi-agent systems (MAS) represent a richer paradigm than actors, so richer seems the technical perspectives

- different, possibly overlapping, declination of the agent notion
  - "weak" agent notion—e.g., JADE [Bellifemine et al., 2007], basically exploiting agent computational autonomy [Mariani et al., 2015]
  - "strong" agent notion—e.g., BDI agents [Bordini et al., 2007, Rao and Georgeff, 1995]
  - richer agent abstractions—e.g., Agents & Artifacts (A&A) meta-model [Omicini et al., 2008]
- basic MAS notions { agent, environment, society } mapped upon BCT & SC
- many diverse mappings are possible



Smart Contracts & MAS II

Example: SC as a BDI agent / Features

- empowering SC with computational autonomy—fitting the distributed scenario
- and goal-oriented computations—enhancing SC expressiveness with goals

Smart Contracts & MAS III

Example: SC as a BDI agent / Issues

- what is the environment, here?
- what can a SC-agent perceive?
- how can goal-oriented reasoning be useful here?
  - should a SC reason about how its business logic?
- what about epistemic actions?
  - should a SC ask for information to other (human?) agents?
- would *multiple intentions* – i.e., multiple control flows – make sensefor a SC-agent?
  - in case, who is paying for them?
  - who is in charge for executing them?
  - and, using which concurrency model?

Smart Contracts & MAS IV

Potential research & technical activities

- exploring all possible mappings
- re-thinking / editing some BCT formal semantics in terms of agents, environment, artefacts, etc.
- re-interpreting existing agent languages (or, proper language subsets) as SC scripting languages

Logic-based Smart Contracts I

Language issues for SC

SCs currently lack

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

Possible lines of development

- investigating how the adoption of a logic interpreted language (e.g., Prolog) may improve SC for the aforementioned issues
- understanding further benefits and possible limitations of logic-based languages for SC—e.g., [Idelberger et al., 2016]

Logic-based Smart Contracts II

Potential benefits

Exploiting a logic language such as Prolog provides some benefits

- SC language becomes 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

Some more questions

- should the computational economic cost model be re-designed to embrace logic programming basic mechanisms?
- how should logic SCs interact?

Potential research & technical activities

- exploiting logic semantics for BCT formal semantics
- exploring the benefits of a declarative language for SC
- experimenting existing languages such as tuProlog [Denti et al., 2005] on top of some existing BCT

More on this on the WOA 2018 talk on the subject!

Blackboard-based Approaches & Smart Contracts

Opportunity

Shared blackboard architectures 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 [Rossi et al., 2001]

e.g. distributed logic programming—e.g., [Ciancarini, 1994, Calegari et al., 2018]

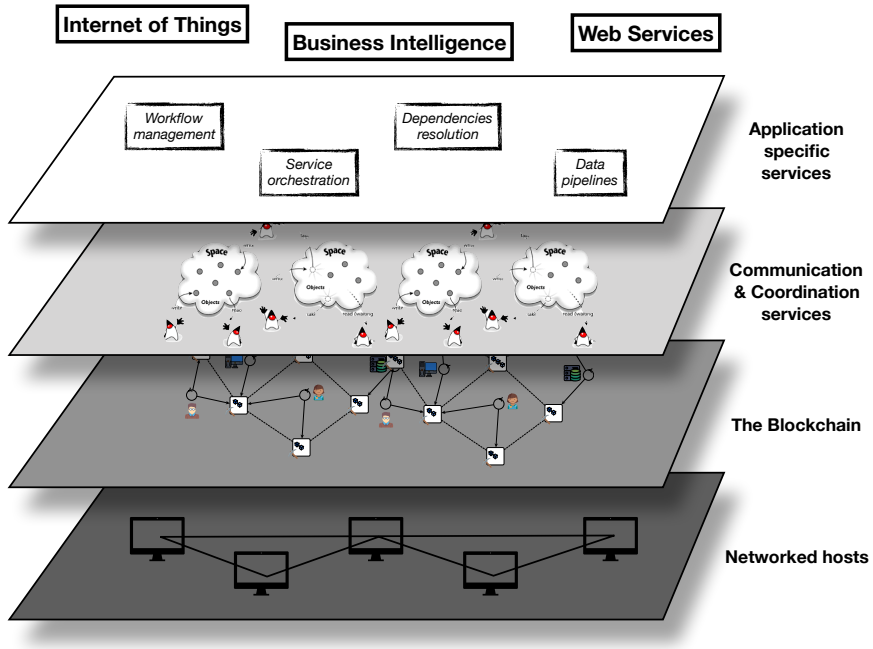
Potential research & technical activities

- determining usefulness of BCTs in such contexts
- in case, using them as application scenarios to devise out new lines for BCT improvement

Tuple-based Coordination upon the Blockchain I

Can we build the archetypal LINDA model [Gelernter, 1985] on top of BCTs?

- if yes, tuple spaces would inherit a lot of desirable properties
  e.g., decentralisation & replication, fault-tolerance, consistency, etc.
- ? which model for economical cost model for LINDA primitives?
- ? how to handle control flow-related aspects?
  e.g., suspensive semantics
- ? can we inject programmability [Denti et al., 1997], too?



Vision: BCT as the backbone on top of which communication and coordination services are built

Tuple-based Coordination upon the Blockchain II

Potential research & technical activities

- comparing different BCTs from the coordination-capabilities point of view, modelling and implementing LINDA on top of them
- comparing different BCTs from the coordination-capabilities point of view, modelling and implementing programmable coordination abstractions [Denti et al., 1997] on top of them—e.g., ReSpecT [Omicini and Denti, 2001]

Distributed LP on the Blockchain I

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 [Shapiro, 1989] has some well-known critic aspects
  e.g., AND-parallelism, OR-parallelism, termination, non-termination, shared variables

? how to handle KB mutability while reasoning?

Potential research & technical activities

- developing a proof of concept or sketched implementation showing the feasibility of concurrent, blockchain-based, SLD resolution process [Robinson, 1965]

Formal (Meta-)Model for BCTs & SCs

Issues

- exception made for Ethereum, all the other mainstream BCTs lack a formal semantics specification
- furthermore, a general meta-model representing and including all BCTs is still unavailable
  - which means that there is no clear consensus on what is the essence of the blockchain notion

Potential research & technical activities

- 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

Simulating the Blockchain

Issue

Some local consensus approaches lack formal theorems proving their properties or their sensibility to the parameters variation

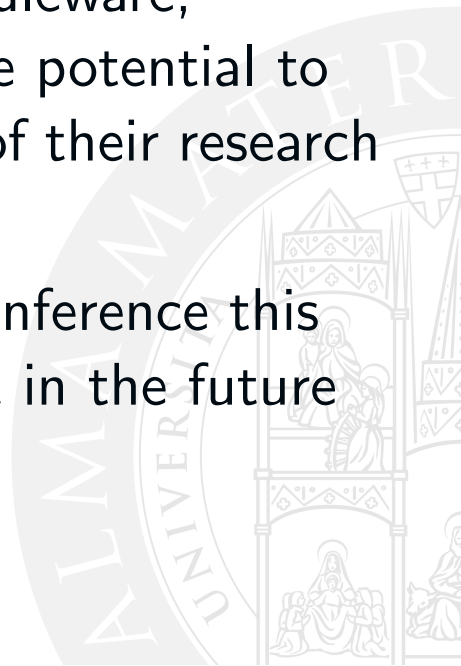
e.g. ΔT, CP distribution, economical cost model, etc.

Possible research goal

- designing & developing an agent-based simulation framework where such interrelated aspects can be studied *in silico*

Just the Beginning

- blockchain and smart contracts bring new features for complex distributed systems—they are much more than new popular buzzwords
- many of the most typical topics of WOA – MAS, middleware, intelligent systems, programming languages – have the potential to successfully exploit BCT and SC in the advancement of their research
- apart from this tutorial and our contribution at the conference this year, we expect to see many contributions of that sort in the future installation of the workshop



References I

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W., and Yellick, J. (2018).
Hyperledger fabric: A distributed operating system for permissioned blockchains.
13th European Conference on Computer Systems (Euro/Sys'18).

Atzei, N., Bartoletti, M., and Cimoli, T. (2017).
A survey of attacks on Ethereum smart contracts (SoK).
In Maffei, M. and Ryan, M., editors, *Principles of Security and Trust*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer.

Aublin, P.-L., Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2015).
The next 700 BFT protocols.
ACM Transactions on Computer Systems, 32(4).

Back, A. (2002).
Hashcash – a denial of service counter-measure.
`http://hashcash.org/papers/hashcash.pdf`.

References II

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007).
Developing Multi-Agent Systems with JADE.
Wiley.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. J. (2007).
Programming Multi-Agent Systems in AgentSpeak using Jason.
John Wiley & Sons, Ltd.
Hardcover.

Brewer, E. A. (2000).
Towards robust distributed systems (abstract).
In *19th Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*, page 7, New York, NY, USA. ACM.

Cachin, C. and Vukolić, M. (2017).
Blockchain consensus protocols in the wild (keynote talk).
In Richa, A. W., editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:16, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

References III


Calegari, R., Denti, E., Mariani, S., and Omicini, A. (2018).
Logic programming as a service.
Theory and Practice of Logic Programming, In press.


Castro, M. and Liskov, B. (2002).
Practical byzantine fault tolerance and proactive recovery.
ACM Transactions on Computer Systems, 20(4):398–461.


Charron-Bost, B., Pedone, F., and Schiper, A., editors (2010).
Replication: Theory and Practice.
Springer.


Chen, L., Xu, L., Shah, N., Gao, Z., Lu, Y., and Shi, W. (2017).
On security analysis of Proof-of-Elapsed-Time (PoET).
In Spirakis, P. and Tsigas, P., editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 10616 of *Lecture Notes in Computer Science*, pages 282–297. Springer.


Ciancarini, P. (1994).
Distributed programming with logic tuple spaces.
New Generation Computing, 12(3):251–283.

References IV

Denti, E., Natali, A., and Omicini, A. (1997).
Programmable coordination media.
In Garlan, D. and Le Métayer, D., editors, *Coordination Languages and Models*, volume 1282 of *LNCS*, pages 274–288. Springer-Verlag.
2nd International Conference (COORDINATION'97), Berlin, Germany, 1–3 September 1997. Proceedings.

Denti, E., Omicini, A., and Ricci, A. (2005).
Multi-paradigm Java-Prolog integration in tuProlog.
Science of Computer Programming, 57(2):217–250.

Douceur, J. R. (2002).
The Sybil attack.
In Druschel, P., Kaashoek, F., and Rowstron, A., editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer.

Dupont, Q. (2017).
Experiments in algorithmic governance. a history and ethnography of "the dao," a failed decentralized autonomous organization.
In Campbell-Verduyn, M., editor, *Bitcoin and Beyond. Cryptocurrencies, Blockchains, and Global Governance*, chapter 8. Routledge, London, UK, 1st edition.

References V

Eyal, I. and Sirer, E. G. (2014).
Majority is not enough: Bitcoin mining is vulnerable.
In Christin, N. and Safavi-Naini, R., editors, *Financial Cryptography and Data Security*, volume 8437 of *Lecture Notes in Computer Science*, pages 436–454. Springer.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985).
Impossibility of distributed consensus with one faulty process.
Journal of the ACM, 32(2):374–382.

Gelernter, D. (1985).
Generative communication in Linda.
ACM Transactions on Programming Languages and Systems, 7(1):80–112.

Haber, S. and Stornetta, W. (1991).
How to time-stamp a digital document.
Journal of Cryptology, 3(2):99–111.

Idelberger, F., Governatori, G., Riveret, R., and Sartor, G. (2016).
Evaluation of logic-based smart contracts for blockchain systems.
In Alferes, J. J., Bertossi, L., Governatori, G., Fodor, P., and Roman, D., editors, *Rule Technologies. Research, Tools, and Applications*, volume 9718 of *Lecture Notes in Computer Science*, pages 167–183. Springer.

References VI

Lamport, L. (1998).
The part-time parliament.
ACM Transactions on Computer Systems, 16(2):133–169.

Lamport, L., Shostak, R., and Pease, M. (1982).
The Byzantine Generals problem.
ACM Transactions on Programming Languages and Systems, 4(3):382–401.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016).
Making smart contracts smarter.
In *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, pages 254–269, New York, NY, USA. ACM Press.

Mariani, S. and Omicini, A. (2013).
TuCSoN on cloud: An event-driven architecture for embodied / disembodied coordination.
In Aversa, R., Kolodzej, J., Zhang, J., Amato, F., and Fortino, G., editors, *Algorithms and Architectures for Parallel Processing*, volume 8286 of *LNCS*, pages 285–294. Springer International Publishing Switzerland.
13th International Conference (ICA3PP-2013), Vietri sul Mare, Italy, 18-20 December 2013. Proceedings, Part II.

References VII

Mariani, S., Omicini, A., and Sangiorgi, L. (2015).
Models of autonomy and coordination: Integrating subjective & objective approaches in agent development frameworks.
In Braubach, L., Camacho, D., Venticinque, S., and Bădică, C., editors, *Intelligent Distributed Computing VIII*, volume 570 of *Studies in Computational Intelligence*, pages 69–79. Springer International Publishing.
8th International Symposium on Intelligent Distributed Computing (IDC 2014), Madrid, Spain, 3-5 September 2014. Proceedings.

Miller, A., Xia, Y., Croman, K., Shi, E., and Song, D. (2016).
The honey badger of BFT protocols.
In *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, pages 31–42.

Nakamoto, S. (2008).
Bitcoin: A peer-to-peer electronic cash system.
http://bitcoin.org/bitcoin.pdf.

Omicini, A. and Denti, E. (2001).
From tuple spaces to tuple centres.
Science of Computer Programming, 41(3):277–294.

References VIII

Omicini, A., Ricci, A., and Viroli, M. (2008).
Artifacts in the A&A meta-model for multi-agent systems.
Autonomous Agents and Multi-Agent Systems, 17(3):432–456.
Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.

Ongaro, D. and Ousterhout, J. (2014).
In search of an understandable consensus algorithm.
In *2014 USENIX Annual Technical Conference (ATC'14)*, pages 305–320.

Popov, S. (2017).
The tangle.
`http://iotatoken.com/IOTA_Whitepaper.pdf`.

Proof-of-Stake (2017).
Proof of stake.
`http://en.bitcoin.it/wiki/Proof_of_Stake`.

Rao, A. S. and Georgeff, M. P. (1995).
BDI agents: From theory to practice.
In Lesser, V. R. and Gasser, L., editors, *1st International Conference on Multi Agent Systems (ICMAS 1995)*, pages 312–319, San Francisco, CA, USA. The MIT Press.

References IX

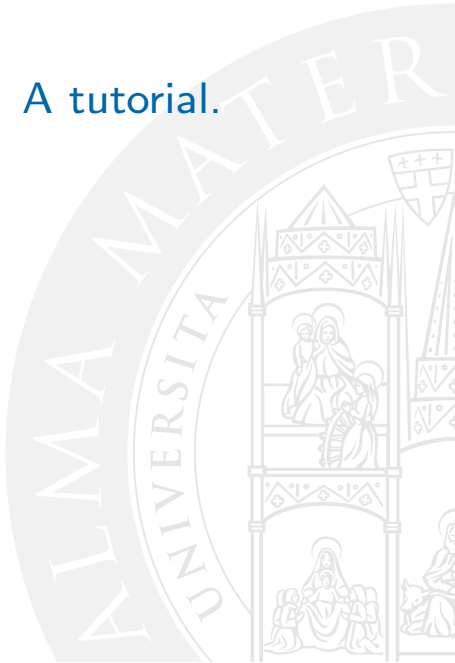
Robinson, J. A. (1965).
A machine-oriented logic based on the resolution principle.
Journal of the ACM, 12(1):23–41.

Rossi, D., Cabri, G., and Denti, E. (2001).
Tuple-based technologies for coordination.
In Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R., editors, *Coordination of Internet Agents. Models, Technologies, and Applications*, chapter 4, pages 83–109. Springer Berlin Heidelberg.

Schneider, F. B. (1990).
Implementing fault-tolerant services using the state machine approach: A tutorial.
ACM Computing Surveys (CSUR), 22(4):299–319.

Seijas, P. L., Thompson, S., and McAdams, D. (2017).
Scripting smart contracts for distributed ledger technology.
Report 2017/1156, IACR Cryptology ePrint Archive.

Shapiro, E. (1989).
The family of concurrent logic programming languages.
ACM Computing Surveys, 21(3):413–510.



References X

Sompolinsky, Y. and Zohar, A. (2013).
Accelerating Bitcoin's transaction processing. fast money grows on trees, not chains.
Report 2013/881, IACR Cryptology ePrint Archive.

Sousa, J. and Bessani, A. (2012).
From Byzantine consensus to BFT state machine replication: A latency-optimal transformation.
9th European Dependable Computing Conference (EDCC 2012), pages 37–48.

Szabo, N. (1996).
Smart contracts: Building blocks for digital markets.
Extropy, 16.

Wood, G. (2014).
Ethereum: a secure decentralised generalised transaction ledger.
Yellow Paper 151, Ethereum Project.

Blockchain & Smart Contracts
Basics and Perspectives for MAS

Andrea Omicini
andrea.omicini@unibo.it

Giovanni Ciatto
giovanni.ciatto@unibo.it

Doctorate Mini-School @ WOA 2018

Palermo, Italy – 27 June 2018