Introduction to the Tezos Blockchain

Victor Allombert*, Mathias Bourgoin* and Julien Tesson*
* Nomadic Labs, Paris, France
Email: firstname.lastname@nomadic-labs.com

INVITED TUTORIAL PAPER

***Abstract*—Tezos is an innovative blockchain that improves on several aspects compared to more established blockchains. It offers an original *proof-of-stake* consensus algorithm and can be used as a decentralized smart contract platform. It has the capacity to amend its own economic protocol through a voting mechanism and focuses on formal methods to improve safety.**

Keywords*— *distributed systems; decentralized systems; blockchains; smart contracts; formal verification.

I. Introduction to Blockchains

At the heart of blockchains lay massively distributed and decentralized programs that aim at bringing consensus (usually over a ledger) among thousands of nodes.

Tezos [1] is an innovative blockchain that improves several aspects compared to more established blockchains like Bitcoin [2] or Ethereum [3]. It offers an original *'Proof-of-Stake'* [4] consensus algorithm and can be used as a decentralized smart contract platform. It has the capacity to amend its own consensus algorithm (and more) through a voting mechanism and focuses on formal methods to improve safety.

Tezos implementation is designed as a multi-layer software. A *peer-to-peer* layer ensures the connectivity with many other nodes and passes the received messages to the next layer that sees this network as a *distributed database*. This layer pulls block chunks from its neighbors, pushes new block identifiers, and passes new information to the *economic protocol* layer which implements the consensus: it decides which blocks with which transactions can be included in the ledger.

The economic protocol also embeds a smart contract platform. Smart contracts are small programs that manage their associated tokens and data storage and that perform blockchain operations. Transactions in the ledger can be more than just mere transfers of tokens (assets residing on the blockchain), they can carry data and can be addressed to a smart-contract. In this case they can be seen as a function call which triggers the execution of the smart-contracts code.

To ensure the consistency of the ledger state, each block received by a node has to be validated before the node agrees to transfer it to its neighbor, thus it is very important for a smart contract execution to be limited in both time and storage in order not to slow down the network.

This tutorial aims at giving a broad presentation of what a blockchain is and how to use and interact with Tezos. We begin with a short introduction to blockchains in the rest of this Section. Section 2 will detail some specifics of Tezos. The next sections will focus on interacting with the Tezos blockchain through multiple examples. First, Section 3 shows how to use the blockchain through a client and *via* a rich set of RPC calls that can be used from any programming language. Then, in Section 4, we present how to use Tezos as a decentralized platform that can run smart contracts. We present through examples how to build and run a small contract while providing some technical details of the Tezos smart contract programming language and platform.

A. Blockchains building blocks

Blockchains can be seen as an immutable database operating in a decentralized network. They are built upon several key concepts and tools:

- They heavily use cryptography to ensure users authentication as well as the database immutability
- They offer a probabilistic solution to the "Byzantine generals problem" [5] for consensus among all participants (that we will call nodes in the rest of this tutorial) in the decentralized network
- They use a peer to peer (P2P) gossip network for low-level communications between the nodes.

Thus blockchains are often called *crypto-ledgers* as they can be seen as an electronic book, recording transactions where users identity and book immutability are cryptographically ensured.

In order to validate and append transactions to the ledger, all blockchains follow a similar generic algorithm:

1) New transactions are broadcasted to all nodes which aggregate them in blocks.
2) The next block is broadcast by one or several nodes.
3) Nodes express their acceptance of a block by including its cryptographic hash in the next block they create.

Blockchains face some common distributed systems challenges. To be resilient to Sybil attacks [6] a solution is to restrict the pool of block producers by tying it to the use of a scarce resource. A difficulty is to choose this resource and create incentives that push the majority of the network to be honest. Restricting the pool of block producers can also lead to liveness issues that have to be taken into account to make sure that the chain does not stop whenever a block producer is offline. Blockchains also have to consider malicious block producers and performance issues such as network delays. Each blockchain provides its own set of solutions to overcome these challenges but most of them rely on the foundations set

978-1-7281-4484-9/19/$31.00 ©2019 IEEE

by Bitcoin. Blockchains also face the risk of forks: to update their economic protocol, blockchains have to go through social consensus and risk frictions in their user community that may lead to the birth of *hard forks* splitting the community in two parts agreeing on two different chains. Such hard fork occurred after the Ethereum DAO hack[1] leading to the birth of two blockchains: Ethereum Classic and Ethereum.

B. Bitcoin: the electronic cash

Bitcoin was introduced in 2008. The main objective behind Bitcoin was to propose a decentralized electronic cash system. Bitcoin is the name of the blockchain as well as the name of the cryptocurrency it uses. Blockchains use tokens to represent assets stored in the chain. Some of these tokens, as in Bitcoin, can represent a currency. Bitcoin is also a decentralized system (the actual blockchain) allowing users to store and transfer their tokens. This system is the combination of a P2P network associated with a consensus protocol to maintain consistency between nodes. This consensus algorithm introduced the notion of 'Proof-of-Work' (PoW). With PoW, the scarce resource used to restrict the pool of block producers (that Bitcoin calls 'miners') is computing power. High computing power mostly demands very efficient computing hardware associated with high energy consumption. The main idea is to request that miners compete in the solving of a puzzle to earn the right to produce (mine) the next block. The puzzle is built to be hard to solve but easy to check. With PoW, Sybil attacks are difficult and expensive. Associated with incentives to motivate miners to compete, liveness is easy to achieve. In case of *soft forks* (temporary split) of the chain, which can be caused by bugs, network latency or malicious mining, and to maintain consistency of the chain among all the nodes, Bitcoin's consensus algorithm specifies to always keep the longest chain between two forks. A chain is considered longer if its total difficulty (summing the difficulty of the puzzles solved to mine each block of the chain) is higher. The difficulty is adapted every two weeks to maintain an average of 10-minute intervals between two consecutive blocks depending on the global computing power of the miners. PoW is the solution currently used by most blockchains.

C. Smart Contracts: decentralized platforms

While Bitcoin focused on electronic cash, it also come with the concept of decentralized computations: Bitcoin's script allows many interesting forms of *in-transaction* computation, and others quickly proposed to use blockchains to build decentralized computing platforms. This was popularized by the current second-biggest blockchain, Ethereum, in 2014, which pushed the concept further. The main idea is to see blockchains as vending machines where users can pay for a service. In blockchain platforms, these services are small programs living on the chain. Users of the blockchain can store code in blocks and other users can execute this code. These programs are called *Smart Contracts*. They can, of course, perform blockchain operations, such as token transfers, but they can also be used for access control or to interact with others. Some smart contracts, being fed with off-chain information, serve as *Oracles* selling trusted information. Smart contracts can be used in many applications such as financial contracts, developing new currencies on top of the basic crypto-currency of the blockchain, voting systems, games or crowdfunding.

[1] https://www.bloomberg.com/features/2017-the-ether-thief/

II. Tezos specifics

Tezos is an innovative blockchain which was presented in 2014. Contrarily to most new blockchains at that time, it is not based on PoW. In the following section, we briefly present some distinctive features of Tezos such as its self-amending mechanism, its usage of a 'Proof-of-Stake' based consensus algorithm and its strong emphasis on formal verification.

A. Self amending blockchain

Tezos is a self-amending crypto-ledger. The protocol that validates blocks and implements the consensus algorithm can amend itself. Concretely a new protocol is downloaded from the network, compiled and how-swapped to replace the current one.

The amendment procedure can be triggered in several ways, depending on the protocol. In the current Tezos economic protocol, an amendment can only be triggered as the result of an on-chain voting procedure.

It helps to avoid forks of the chain and reduces friction and splitting in the community. A protocol amendment may consist of very important upgrades such as a switch to a completely different consensus protocol. It can also consist in smaller modifications such as extending the smart contract language, modifying the rewarding system to enforce network participation or adding new kinds of transactions (for instance adding anonymous ones). In order to amend itself, Tezos uses an on-chain voting system where users of the blockchain participate to propose, select, adopt or reject new amendments. The voting process is currently divided in 4 periods of ~3 weeks (time is measured in blocks, Tezos aiming at a 1-minute intervals between two consecutive blocks):

1) Participants submit new protocol proposals (i.e. hashes of the protocol proposals source files)
2) A first vote selects a proposal among the submitted proposal
3) A side test chain spawns with the elected protocol
4) A final vote occurs that decides whether to upgrade(needing a supermajority of 80% of positive votes)

Tezos is built specifically in order to be self-amendable. Tezos nodes are split into two parts: the *protocol*, which is the self amendable part, and which is isolated from the *shell* that is responsible for the low-level network communications.

The shell can be written in any programming language. There can be multiple implementations with different properties. It corresponds to the first two layers of Tezos architecture.

The protocol has to run exactly the same way on all nodes. It is responsible for validating blocks and operations. Operations,

that are aggregated into blocks, are what is stored in the ledger and are of two kinds:

- ledger operations: transactions and origination (creation) of contracts
- Proof-of-Stake operations such as endorsements or delegation that are described in Section 2.2.

The protocol can also trigger a protocol upgrade. In order to allow all kinds of protocols to be compatible with the shell, Tezos reduces the protocol interface as much as possible. In Tezos, the generic operations of regular blockchains are implemented as a purely functional module. Thus, well known blockchains such as Bitcoin or Ethereum can all be represented within Tezos by implementing the correct interface to the shell. The interface of the protocol is primarily composed of two functions: **apply** and **score**.

$$\begin{aligned} \textbf{apply} &: S \times B \to S \\ \textbf{score} &: S \to \mathbb{N} \end{aligned}$$

where S, the *state* of the blockchain (the ledger), is an immutable key-value store and B is a block. **apply** takes the current state and a block to produce a new state. **score** computes the score of a state to choose the preferred one between multiple states (implemented as the longest chain in Bitcoin). A few other functions are exposed for efficiency purposes, to document errors and provide protocol-dependent RPCs.

Tezos protocols are written in the OCaml programming language [7]. OCaml is a powerful functional programming language offering speed, an unambiguous syntax and semantic, and a rich ecosystem that makes Tezos a good candidate for formal proofs of correctness. To make it more resilient and less error-prone, the protocol has only restricted access to the standard library: for instance it uses no I/O functions, no threads, no unsafe language traits. It also has access to specific libraries such as formally verified cryptographic libraries or database abstractions.

B. 'Proof-of-stake' based consensus algorithm

The current Tezos protocol is based on a 'Liquid Proof-of-Stake' (LPoS) consensus algorithm.

PoS is very different from PoW. It considers the stake (the number of tokens) that users hold as the primary resource used to build the pool of block producers (called *bakers* in the Tezos ecosystem). In the current Tezos consensus protocol, to push a block at a certain level, bakers are randomly selected using a lazy infinite priority list of baking slots. In order to participate in this random selection, a baker must hold at least a roll of tokens (corresponding to $10,000$ tokens[2]). The number of baking slots is proportional to the number of rolls that a baker holds. However, participants that do not hold enough tokens or who do not wish to bake blocks can delegate their tokens to another baker, much like in Liquid Democracy one can delegate its right to vote. They keep the ownership of their tokens but increase the stake of their delegate in the random assignment of baking slots. Delegation makes the PoS system more fair and participative and helps balance a possible concentration of tokens in few hands.

In order to help the chain reach finality (the guarantee that a block will not be revoked and that past transactions can never change) faster, Tezos PoS mechanism introduces endorsements of baked blocks. For each baked block, 32 endorsements (signatures) slots are created, allowing chosen bakers to approve a block by signing it. Using endorsements, the highest block resulting score is considered the head of the chain where the score is:

$$\textbf{score}(B_{n+1}) \quad = \quad 1 + \textbf{score}(B_n) + \text{nb\_endorsements}$$

In order to provide incentives to bakers for participating in the network, the protocol rewards baking and endorsing. A baker earns 16 tokens for each block it bakes and 2 tokens for each endorsement it produces.

Two main malicious behaviors are also handled by the protocol: *double baking* and *double endorsement*. A baker perpetrates double baking when it injects two different blocks at the same level. Double endorsements happens when a baker signs two different blocks for the same level. The system punishes malicious behaviors as follows: when a baker produces a block, a deposit bond of 256 tokens is frozen for ~2 weeks (64 tokens for an endorsement). During this period, if the baker/endorser is caught cheating, the deposit and pending rewards (summing the rewards earned baking and endorsing in the last 5 cycles $-$ a cycle equals 4096 blocks) of the cheater are forfeited.

Tezos LPoS consensus algorithm, *via* its internal mechanism and its associated incentives, solves the challenges presented in Section 1 without requiring significant computational power. It also focuses on the users of the platform (instead of external actors as it is possible with PoW): only stake-holders can participate but all of them can, from large ones with many rolls to smaller ones that delegate their stake.

C. Strong emphasis on formal verification

Tezos uses as much as possible state-of-the-art programming languages capacities to statically ensure the correctness of the implementation and limit the possible runtime errors or attacks.

The code base is mainly written in the OCaml programming language, whose robust static type system and memory management system rule out many common runtime errors like null pointer exceptions or buffer overflows.

Regarding cryptographic primitives implementation, whose importance in terms of security is paramount for the blockchain, Tezos relies on the HACL* library [8] which is implemented in Low* [9] and extracted to C. The cryptographic primitive implementation is formally proven to be *memory safe*, *functionally correct* and resistant to *side-channel attacks*

[2] $10,000$ tokens while writing these lines. An amendment of the protocol, currently in the testing phase of the voting process described in the previous subsection, reduces the size of a roll to $8,000$ tokens

at least at the level of C (secret independence of branching and memory access).

Michelson, the Tezos smart contract language, has been explicitly designed to ease the readability and verifiability of contracts while being low level enough to comply with the performance predictability requirement of on-chain execution. The language is statically typed, its formal semantics has been written in the Coq proof assistant [10] and formal proofs of functional correctness of smart contract using this semantics have been done.

III. Interacting with the Tezos Blockchain

The architecture of Tezos is centered around two main components.

First, the *node* (the corresponding executable file being called `tezos-node`) is responsible for connecting to peers through the gossip network and updating the ledger's state (*context*). As all the blocks and operations are exchanged between nodes on the gossip network, the node is able to filter and propagate data from/to its connected peers. Using the blocks received on the gossip network, the node keeps an up-to-date context. The node can be run with several daemons such as `tezos-baker-*` and `tezos-endorser-*` which take part of the consensus algorithm by, respectively, baking and endorsing blocks.

Second, the *client* (`tezos-client`) is the main tool to interact with a Tezos blockchain node.

There are currently 3 public Tezos networks:

- `mainnet` which is the current incarnation of the Tezos blockchain. It runs with real tez (Tezos tokens) that have been baked or allocated to the donors of July 2017 fundraiser. It has been live and open since June 30th 2018
- `alphanet` which is based on the `mainnet` code base but uses free tokens. It is the reference network for developers wanting to test their software.
- `zeronet` wich is the testing network, with free tokens and frequent resets.

In this tutorial, we will use the `alphanet` Tezos network.

In the following sections, we assume that the reader have access to the Tezos binaries. A pre-configured Tezos environment can be found in the provided virtrual machine. Otherwise, it is possible to install a Tezos environment from source (using the ocaml package manager (opam) and compiling from source) or with docker (using scripts and images). All the instructions to install and run the Tezos software from source or from docker can be found at http://tezos.gitlab.io/master/introduction/howtoget.html.

A. Seting up a Tezos node (*demo*)

The *node* (`tezos-node`) can be considered as the *access point* to the Tezos blockchain and stores all the data necessary to run the blockchain. In practice, the node's data is stored (by default) into the `~/.tezos-node` directory.

To be connected to the network, a node must have a proper network identity to be globally identified.

To generate an identity, the following command should be run:

```
tezos-node identity generate
```

The generated identity will be stored as a pair of cryptographic keys that are used by the node to send encrypted messages, but it is also used as an antispam measure (to prevent Sybil attacks) based on a lightweight PoW.

When the identity is generated, we can run a node using:

```
tezos-node run --rpc-addr 127.0.0.1
```

The `-rpc-addr 127.0.0.1` argument is used to allow communications with clients on the local host only. The node is now able to connect to the Tezos network and will start its *bootstrap* phase. It consists in downloading all the blocks from the chain using the distributed network. This procedure can be very long as the chain data is growing invariably every day. To speedup the process (from days to minutes), it is possible to start a node from a snapshot of the chain[3] by running:

```
tezos-node snapshot import last.full
```

This command is able to read all the necessary data stored in the `last.full` file, validate it and import it in the node storage. The imported data consist in a partial ledger state (that can be reconstructed on request) and all the blocks of the chain since the genesis. It is also possible to set up a lightweight node targeting low resource architectures by running a partial chain using a *rolling* snapshot. When the import is done, one can run the node, and wait a few minutes to download the new blocks spawned since the snapshot file was exported.

B. Using basic client commands (*demo*)

The *client* (**`tezos-client`**) is a user-friendly interface that can be used to interact with a node. As it is based on JSON RPCs, it can be requested by various third-party applications. For the sake of brevity, we will use **`t-c`** instead of **`tezos-client`** in the rest of the document.

The client can be used to check if the current head of the local node is up-to-date using **`t-c bootstrapped`**. This command will hang and return only when the node is synchronised.

The client is also able to handle a simple wallet, stored (by default) in the `~/.t-c` directory. It mainly contains 3 files : `public_keys`, `secret_keys` and `public_key_hashes` (Tezos addresses : *tz1*). To generate a new pair of keys to be used locally for **Bob**, we can run:

```
t-c gen keys bob
```

In order to test the network and help users get familiar with the system, you can obtain free tokens from a faucet[4]: https://faucet.tzalpha.net/. This service will provide a simple wallet formatted as a JSON file. The account can be activated for an identity using:

[3]To avoid to download a fake chain, it is necessary to carefully check that the block hash of the imported block is included in the chain. However, we do not detail the procedure here.

[4]Please drink carefully and don't abuse the faucet: it only contains 30,000 wallets for a total amount of 760,000,000 tokens.

```
t-c activate account alice with "
↪   tz1__xxxxxxxxx__.json"
```

We can now check the balance of this account using:

```
t-c get balance for alice
```

It is time to try to transfer some tokens from one account to another. To transfer 1 token from Alice's account to Bob's one, we can run

```
t-c transfer 1 from alice to bob --fee 0.05
```

The -fee argument stands for the fees associated to an operation in order to encourage bakers to include our operation in a block. To be sure that the operation is well included in the chain, it is advised to wait 60 blocks (~60 min) to consider it as a *valid transaction* using

```
t-c wait for <operation hash> to be included
```

Client commands are high-level operations implemented using the set of RPCs exposed by the Tezos node. The next section presents how the transfer operation can be implemented manually using some of these RPCs.

C. Using RPCs

In this section, we show how to transfer tokens from one account to another using RPCs. We will use the client to call the RPCs of the associated node.

The whole set of RPCs can be found in the *JSON/RPC interface* section of the online Tezos documentation [11] or using the following client command:

```
t-c rpc list
```

The -l option of the client logs all the requests to the node. The following command shows all the RPC calls made during a transfer.

```
t-c -l transfer 1 from bob to alice --fee
↪   0.05
```

As we can see, it consists of 20 calls to the node.

In this tutorial, we will only focus on the 10 mandatory calls to make a transfer. For readability we will use some shortcuts.

- BOB corresponds to Bob's public key.
- ALICE corresponds to Alice's public key.
- HEAD_HASH corresponds to the hash of the head block.
- CHAIN_ID corresponds to the id (hash) of the chain.

1) In Tezos, account operations are numbered, in order to prevent replay attacks. Nodes can be queried to get the current counter and compute a new counter (by incrementing the current one) to forge a new operation. If the new operation has an incorrect counter, it can be ignored, or delayed. The following command gives the current counter for Bob's account.

```
t-c rpc get
 /chains/main/blocks/head/context/
↪   contracts/BOB/counter
```

2) For signature check of the incoming transaction, it is mandatory to verify that the sender public key is known on the blockchain.

```
t-c rpc get
   /chains/main/blocks/head/context/
↪   contracts/BOB/manager_key
```

3) To make sure that the transaction will be added into the blockchain, we have to make sure that the node is bootstrapped (ie. that it is synchronized with the other nodes in the system).

```
t-c rpc get /monitor/bootstrapped
```

4) Some values have to be given to the transaction operation, in particular the gas_limit and storage_limit (see Sec IV-A and IV-B) constants can be queried :

```
t-c rpc get
/chains/main/blocks/head/context/
↪   constants
```

The needed information can be extracted from the JSON answer:

```
{ ...
 "hard_gas_limit_per_operation": "
 ↪   400000",
 ...
 "hard_storage_limit_per_operation": "
 ↪   60000" }
```

5) The hash of the head is also needed:

```
t-c rpc get /chains/main/blocks/head/
↪   hash
```

6) As well as the id of the chain:

```
t-c rpc get /chains/main/chain_id
```

7) We can now simulate the execution of our operation:

```
t-c rpc post /chains/main/blocks/head/
↪   helpers/scripts/run_operation
```

Here we use a POST that demands a JSON input.

```
{ "branch": "HEAD_HASH",
  "contents":
  [ { "kind": "transaction",
    "source": "BOB",
    "fee": "50000",
    "counter": "4",
    "gas_limit": "400000",
    "storage_limit": "60000",
    "amount": "1000000",
    "destination": "ALICE" } ],
  "signature": ANY_SIGNATURE ... }
```

We can use ANY_SIGNATURE to make the simulation without signature checks. In the JSON answer, we get how much gas and storage were consumed. {... "
↪   consumed_gas": "10100"...}

8) We can now adjust the fees, gas limit and storage limit based on last RPC result and run the simulation with signature check.

```
t-c rpc post
/chains/main/blocks/head/helpers/
↪   preapply/operations
```

```
[ { "protocol": "
↪   ProtoALphaALphaALphaALphaALp...",
  "branch": "HEAD_HASH",
  "contents":
  [ { "kind": "transaction",
    "source": "BOB",
    "fee": "1269",
    "counter": "1",
    "gas_limit": "10200",
    "storage_limit": "0",
    "amount": "1",
    "destination": "ALICE" } ],
  "signature": "edsigtf12Ls...}␣]
```

9) We can now inject the operation:

```
t-c rpc post
injection/operation?chain=main
```

This RPC call take an hex-encoded signed operation as input (`"09115800..."`) and returns an operation identifier (`"opDerPd..."`).

10) An additional `POST` RPC call (that is not used by the client) can be helpful to compute the hex-encoded operation:

```
t-c rpc post
/chains/main/blocks/head/helpers/forge
↪   /operations
```

IV. Tezos as a decentralized platform

As mentioned before, Tezos economic protocol not only handles a registry of transactions, but also has support for smart contracts.

Smart contracts are small programs registered in the blockchain together with a private data storage: meaning that only the contract can interact with the storage, but the data are publicly visible. A contract registered in the chain is said to be *originated* and it has an address prefixed by KT1 which is given in the contract's origination block.

They are executed by performing specific transactions to their associated account. The transaction carries data that are passed as a program parameters and can thus be viewed as a procedure call.

The execution of a smart contract can change the state of its storage and trigger on chain transactions.

Smart contract languages are usually Turing-complete. However the replicated nature of the contract storage and the liveliness requirement of the consensus algorithm imposes some restrictions on their execution.

A. Limited execution time

Any call to a smart contract, once included in a block, will be executed on every node in the P2P network, because they have to validate the block before including it in their view of the chain and before passing it to their neighbors. That means that the execution time of each smart contract call included in a block has to fit multiple times in the inter-block time of the chain (1 minute for Tezos) to ensure its liveliness.

Thus each call is allowed a bounded quantity of computation: the smart contracts interpreter uses the concept of *gas*. Each low-level instruction evaluation burns an amount of gas which is crafted to be proportional to the actual execution time and if an execution exceeds its allowed gas consumption, it is stopped immediately and the effects of the execution are rolled back. The transaction is still included in the block and the fees are taken, to prevent the nodes from being spammed with failing transactions.

In Tezos, the economic protocol sets the gas limit per block and for each transaction, and the emitter of the transaction also set an upper bound to the gas consumption for its transaction. The economic protocol does not require the transaction fee to be proportional to the gas upper bound, however the default strategy of the baking software (that forges blocks) provided with Tezos current implementation does require it.

B. Data storage

Each smart contract on the chain possesses its own storage, only accessible to the contract. As this storage is replicated on every node that runs the chain, it has to be of limited size in order to avoid that the chain context grows out of control. A cost is set for storage allocation (currently 0.001tez per byte) to restrain storage usage.

C. Michelson: Tezos' smart-contract programming language

1) Design rationale: The constrained context in which smart contracts operate imposes strong contradicting constraints on the language design.

Because we need to be able to accurately account for resources consumption, the language has to be interpreted. The interpreter is thus counting gas at each "opcode", and each opcode cost has to be fairly simple to guess. This tends to push to a low-level language, at the same time, however, the resource constraint will lead people to write their program in this language. Indeed, they do not want to rely on a very high-level language with a compiler performing many under-the-hood transformations, preventing cost predictions. Therefore, the language has to be high-level enough to be programmable by a human.

Furthermore, as the program will be stored on chain in this language, it is of paramount importance that they can be audited easily. The language has to be simple, high-level enough and should offers as few means of code obfuscation as possible in order not to mislead the reader.

One more constraint is that the language gives as much guarantees as possible statically, as once published on the chain, it is not possible to modify it to correct bugs anymore. So we want to have a strong type system that prevents as much runtime error as possible.

2) A stack language with high-level data structures: Michelson, the smart contract language on Tezos is a stack based language *à la* Forth with strict static type checking and high-level data structures *à la* ML.

A Michelson program is a sequence of instructions which modify the stack given as a program input. The initial stack contains only the calling argument and the contract's storage,

and the program must end with a stack containing only a list of operations paired with the new value of the storage.

This led to a rather simple interpreter, with simple cost model for most operations, but with high-level data structures (such as maps, sets, lists and algebraic data types) to help the writing of smart contracts.

The operations – i.e. Tezos transactions (including calls to other contracts), contract creations and delegate setting – will only be executed after the program returns. This prevents reentrancy bugs (which are hard to spot and have costs millions and provoked the hard fork on Ethereum after the DAO attack). We will discuss contract interaction with an example hereinafter (IV-C4).

The type of each instruction describes the states of the stack before and after the instruction. For example, the instruction DUP has type `'a:'S →'a:'a:'S` meaning that when starting from a stack whose top element has type `int`, the duplication of the top element leads to a stack with one more element of type `int` on top of it (i.e. `int:int:'S`). Thus the type checking of the contracts ensure that no instruction can failed because of a malformed stack.

While the type of the Michelson instruction is polymorphic, the type of contracts arguments and storage have to be monomorphic. This is partly to keep the type checking simple enough to be done efficiently: contract type checking consumes gas and has to be efficient.

Rather than going into the details of all the language instructions, we will provide here two programs examples. The interested reader can find the full description of the language in the Tezos documentation [12].

3) A voting contract: As a first example, we will describe a voting contract. The use cases for such a contract range from voting for your favourite supercomputer in a TV show, to registering vote for important decisions in a decentralized infrastructure. The first one is a bit simpler to implement than the second as we don't have to check the identities of the voters, so we will focus on this: an open vote with a fee, to determine the preference of the voters in a fixed list of choices.

In the following lines, we will present an abstraction of the state of the stack with a comment (prefixed by #) after each relevant block of code.

We start with a storage which holds the names that voters are allowed to vote for, associated with the number of votes they received. Thus we start our program by declaring the type of the storage: a map from `string` to `nat`.

```
storage (map string int);
```

Then we specify the type of the parameter:

```
parameter string;
```

and now we can write the code of the contract. The contract execution starts with the parameter paired on top of an empty stack:

```
code{
  # (name,storage)
```

First we verify that the caller send us enough token to be able to vote. If not, we make the call fail.

```
  AMOUNT;
  # amount:(name,storage)
  PUSH mutez 5000 ;
  # 5000:amount:(name,storage)
  IFCMPGT{PUSH string "stingy␣!" ;
      FAILWITH}
        {};
  # (name,storage)
```

AMOUNT pushes on the stack the number of tokens received from the contract caller, `PUSH 'a cst` pushes the given constant `cst` of type `'a` on the stack. IFCMPGT is a macro which compares the two numbers on top of the stack (removing them in the process) and if the first element was greater it executes its first parameter, otherwise the second.

If payment is sufficient, we prepare the stack by duplicating the pair holding the parameter and the storage, the first (name, map) pair will be consumed by GET to obtain the current number of votes, while the second will be used to produce the new map.

```
  DUP;
  # (name,storage):(name,storage)
```

To get the value of interest from the storage we first destruct the pair to get a stack with the key on top and the map beneath, and then we apply the GET instruction.

```
  UNPAIR;GET;
  # (Some current | None):(name,storage)
```

We get the current count for the voted name or `None` if the key was not in the map. If the count is some integer value, we add 1 to this value, if not we fail because the vote is for an unknown name.

```
  IF_SOME
    {PUSH int 1;ADD;SOME}
    {PUSH string "Unknown␣supercomputer";
       FAILWITH};
  # (Some current+1):(name,storage)
```

We now reorder the elements to prepare the stack to use UPDATE in order to update the map with the new count. DIP allows to work on the element below the stack top, SWAP exchanges the two top elements of the stack.

```
  DIP{UNPAIR}; SWAP;
  # name:(Some current+1):storage
  UPDATE;
  # updated storage
```

We get a new storage, that we pair with an empty list of operations to match the return type of the contract, a pair (list of operations, storage).

```
  NIL operation ; PAIR
  # (nil, updated storage)
}
```

This rather simple program can be extended in many ways. For example, a deadline for the vote could be fixed by storing the end date in the storage and comparing it to the value pushed by NOW on the stack. Or we could grant the right to add new names to the map for voters transferring a bigger amount to the contract. Finally, we could store the addresses of voters (obtained with the instruction SENDER) and reward the voters who voted for the winner.

All these improvements are left as exercises for the interested reader.

4) Inter-contract calls: As stated before, to prevent reentrancy bugs, calls to other contracts are performed at the end of the current contract execution, thus to emulate a procedure call expecting a return data, inter-contract calls will have to use callbacks.

Let us take as an example an insurance contract A, which calls a meteorological oracle contract B, with a given parameter like a date to obtain a related data (say the hydrometry of the given date). The contract A will pass to B a callback identifier, like the insured address, B will now call A with the relevant data and the callback identifier. The contract A can now proceed to the refunding of the legitimate contracting party depending on the data received from the oracle.

We start with a simple oracle which is a contract that just encapsulates a map but guarantees that only a trusted source – whose address is in the contract storage – can modify the map.

For the sake of conciseness, we omit in this contract the usual code allowing the oracle manager, whose key is stored in the contract, to withdraw the tokens held by the contract. As contract will probably be non-spendable in the future, meaning the owner of the contract will not be able to transfer tokens of the contract, only code will, this would freeze the tokens associated to the contract.

The parameter given to the contract is either a new key-data pair to be updated, or a request for the data matching a given key.

```
parameter (or (timestamp :lookup_key)
              (pair
                 (timestamp :lookup_key)
                 (nat :rain_level)));
```

and the storage is a map together with the manager key:

```
storage ( pair
            (big_map
               (timestamp :lookup_key)
               (nat :rain_level))
            (address :oracle_manager) ) ;
```

The code separates the parameter from the storage and then checks with IF_RIGHT whether the call is an update (right of the or type) or a client query (left of the or).

```
code {
 UNPAIR;
 # parameter:(map,oracle_addr)
 IF_RIGHT { # feeding the oracle with data
 # (timestamp,level):(map,oracle_addr)
```

For data update, we first check that the data are indeed provided by the registered manager, and fail if it is not the case:

```
  DIP {
    UNPAIR;SWAP;DUP;SENDER ;
      # sender:oracle_addr:oracle_addr:map
      ASSERT_CMPEQ; SWAP
      # map:oracle_addr
    };
    # (timestamp,level):map:oracle_addr
```

The map is then updated with the provided values:

```
    UNPAIR; DIP{SOME};UPDATE;
    # map:oracle_addr
    PAIR ; NIL operation ; PAIR
  }
```

If the call is a request from a client we retrieve the data from the map and then craft a call to the sender with this value as parameter.

```
  {
    # Getting the data
    DIP{DUP;CAR};GET;
    # (Some level|None):(map,oracle_addr)
    LEFT unit;
    # (Left (Some level|None)):(map,
        oracle_addr)
    # preparing the reply to the caller
    DIP {
      SENDER;
      CONTRACT (or
                (option(nat :rain_level))
                (unit));
      ASSERT_SOME;
      PUSH mutez 0
        };
    # param:0:sender_ctrct:(map,oracle_addr
        )
    TRANSFER_TOKENS ;
    NIL operation; SWAP;CONS;PAIR
  }
};
```

We can now define our insurance contract. Its parameter type has to be the type expected by the oracle (The one used in the oracle transfer). In this small example it is easy, but for more general-purpose oracle interacting with more general purpose client contract, the later won't be able to have all the same type, so the usual mechanism is to originate a proxy contract whose type satisfies the oracle requirement and which can relay the calls between the oracle and the client contract.

We will assume here that the insurance contract is issued for a one-time insurance: given a rain level threshold at a certain

point in time, it will redeem one or the other of the registered addresses of the contracting parties.

The contract can be called by anyone with `Right Unit` to trigger the redeeming mechanism or by the oracle with `Left (Some level)` (callback).

```
parameter
  (or (option(nat :rain_level)) unit);
```

The storage holds the contract parameters: timestamp at which the rain level should be checked, rain level threshold, redeeming addresses and address of the oracle to consult.

```
storage
  (pair
    (pair
      timestamp
      (pair
        (pair
          (contract %under_key unit)
          (contract %over_key unit))
        (nat :rain_level %threshold)))
    (contract %oracle_contract
      (or
        (timestamp :lookup_key)
        (pair (timestamp :lookup_key) (
           nat :rain_level))))
  );
```

The code inspects the given parameter, if the parameter is `Left (Some level)` then we first check that the sender is indeed the oracle and then proceed to the redeeming:

```
code
  { UNPAIR;
    IF_LEFT # callback
      { DIP{DUP;CDR;ADDRESS;SENDER;
          ASSERT_CMPEQ};
        # OK it comes from the Oracle
        ASSERT_SOME ;
        #Ok the oracle has data for the
            timestamp
        DIP{DUP;CAR;CDR;UNPAIR;SWAP};
        IFCMPLT {CAR %under_key} {CDR %
            over_key};
        # We selected contract which receive
             tokens
        BALANCE; UNIT ; TRANSFER_TOKENS;
        # Setup the transfer, then rework
            the stack to satisfy the return
            type
        NIL operation ; SWAP ; CONS ;
        PAIR
      }
```

Else the call triggers the call to the oracle.

```
      { DROP; #dropping Unit
        DUP;UNPAIR;CAR; #getting key
        LEFT (pair (timestamp :lookup_key)
            (nat :rain_level));
        DIP{PUSH mutez 1000}; # pushing the
             fee for the Oracle
        TRANSFER_TOKENS; #calling the
            oracle
        NIL operation; SWAP; CONS; PAIR
      }
  }
```

D. Contract origination and call

To originate the voting contract for a vote on your favorite supercomputer, we can use the Alice account with the following command:

```
t-c originate contract vote\
  for alice transferring 0 from alice \
  running ./vote.tz \
  --init\
  '{ Elt "Sierra" 0 ; Elt "Summit" 0 ;
   Elt "Sunway" 0 ; Elt "Tianhe-2A" 0 }'\
  --burn-cap 1
```

The elements of the storage have to be in alphabetical order.

Then we can vote for Summit, using the following transaction:

```
t-c transfer 0.005 from bob to vote\
  --arg '"Summit"' --burn-cap 1
```

If we issue the transaction with a 0.001 instead of 0.005, the transaction will fail, so we will keep the 0.001tez but we will loose the fees for the baker.

To test our Oracle/insurance example, we first originate the oracle, as we need to initialise the insurance storage with the KT1 address of the oracle. This address is generated when the contract is originated.

The initial storage use the address of Alice, tz1_XXXX, as oracle manager, meaning that only transactions initiated by Alice can add data to the contracts map.

```
t-c originate contract oracle\
  for alice transferring 0 from alice\
  running ./oracle_ok.tz\
  --init 'Pair { } "tz1_XXXX" ' --burn-cap 1
```

The origination receipt gives us the contract address, or we can retrieve it later with the client:

```
t-c show known contract oracle
```

Let say the address of the contract is KT1_YYYY, we now can originate our insurance:

```
t-c originate contract insurance \
  for bob transferring 100 from bob\
  running ./insurance.tz --init\
  'Pair
    (Pair "2019-05-07␣23:22:25+00:00"
      (Pair (Pair "tz1_AAAA" "tz1_BBBB") 10))
    KT1_YYYY'
```

Alice account can feed the oracle contract with data:

```
t-c transfer 0 from bootstrap1 to oracle
  --arg 'Right
    (Pair "2019-05-07␣23:22:25+00:00" 15)'
```

and we can check that the data is indeed in the storage of the contract by inspecting it:

```
t-c get script storage for oracle
```

Finally anyone can trigger our insurance:

```
t-c transfer 0 from charlie to insurance
  --arg 'Right Unit' --burn-cap 1
```

The receipt shows that:

- the insurance contract makes a transfer to the oracle
- the oracle makes a transfer back to the insurance contract
- the insurance contract makes a transfer to the registered contracting address

GLOSSARY

Baker: entity responsible for selecting operations to produce a block in Tezos
Block: set of operations, aggregated in the blockchain
Blockchain: distributed database formed as a list of blocks
Client: entity responsible for interacting with a node
Context: Ledger's state (accounts balance, contracts, ...)
Cycle: set of consecutive blocks
Delegate: entity to which an account has delegated stake
Endorser: seal of approval for a block
Liveness: mandatory property allowing the system to progress
Miner: entity responsible for selecting operations to produce a block
Node: entity responsible for connecting to a Tezos network
Operation: transforms the context
Oracle: off-chain third party that can deliver data
Origination: operation to create an account that can contains a contract or be delegated
PoS: Proof-of-Stake
Pow: Proof-of-Work
Roll: amount of tokens used to determine delegates' rights
RPC: Remote Procedure Call
Self-amending: ability to update itself seamlessly
Smart contract: originated account which is associated to a Michelson script
Stake: amount of token
Storage: blockchain data necessary to run a node
Sybil attack: take over the network by flooding malicious identities
Token: unit of value
Tz1: Tezos implicit account address
KT1: Tezos originated account address

REFERENCES

[1] L. Goodman, "Tezos – a self-amending crypto-ledger," https://www.tezos.com/static/papers/white_paper.pdf, 2014.
[2] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," https://bitcoin.com/bitcoin.pdf, 2008.
[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," http://gavwood.com/paper.pdf, 2014.
[4] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," *self-published paper, August*, vol. 19, 2012.
[5] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.
[6] J. R. Douceur, "The Sybil Attack," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 251–260.
[7] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The OCaml system release 4.07: Documentation and user's manual," Inria, Intern report, Jul. 2018. [Online]. Available: https://hal.inria.fr/hal-00930213
[8] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," Cryptology ePrint Archive, Report 2017/536, 2017, https://eprint.iacr.org/2017/536.
[9] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, C. Hritcu, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoué, "Verified low-level programming embedded in F*," *CoRR*, vol. abs/1703.00053, 2017. [Online]. Available: http://arxiv.org/abs/1703.00053
[10] The Coq Development Team, "The Coq Proof Assistant," http://coq.inria.fr.
[11] "The Tezos Developer Resources," http://tezos.gitlab.io/master/.
[12] The Tezos Development Team, "Michelson Reference Manual," http://tezos.gitlab.io/master/whitedoc/michelson.html.