CST 428 BLOCK CHAIN TECHNOLOGIES

S8 CSE – ELECTIVE

MODULE – 5

Ethereum and Solidity

Ethereum – an overview

- idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and Decentralized Applications (DApps)
- first version of Ethereum, called Olympic, was released in May, 2015
- Two months later, Frontier was released in July

Ethereum – an overview

- Another version named Homestead with various improvements was released in March, 2016
- The latest Ethereum release is called Muir Glacier

The Ethereum network

- a peer-to-peer network where nodes participate in order to maintain the blockchain and contribute to the consensus mechanism
- Networks can be divided into three types, based on the requirements and usage
- The mainnet - current live network of Ethereum with network ID and chain ID as 1 to identify the network

The Ethereum network

- Testnets - to provide a testing environment for smart contracts and DApps before being deployed to the production live blockchain
    - Also allow experimentation and research
    - Example - Ropsten, Kovan and Rinkeby,

The Ethereum network

- Private nets - private networks that can be created by generating a new genesis block in private blockchain networks, where a private group of entities start their blockchain network and use it as a permissioned or consortium blockchain

Components of the Ethereum ecosystem

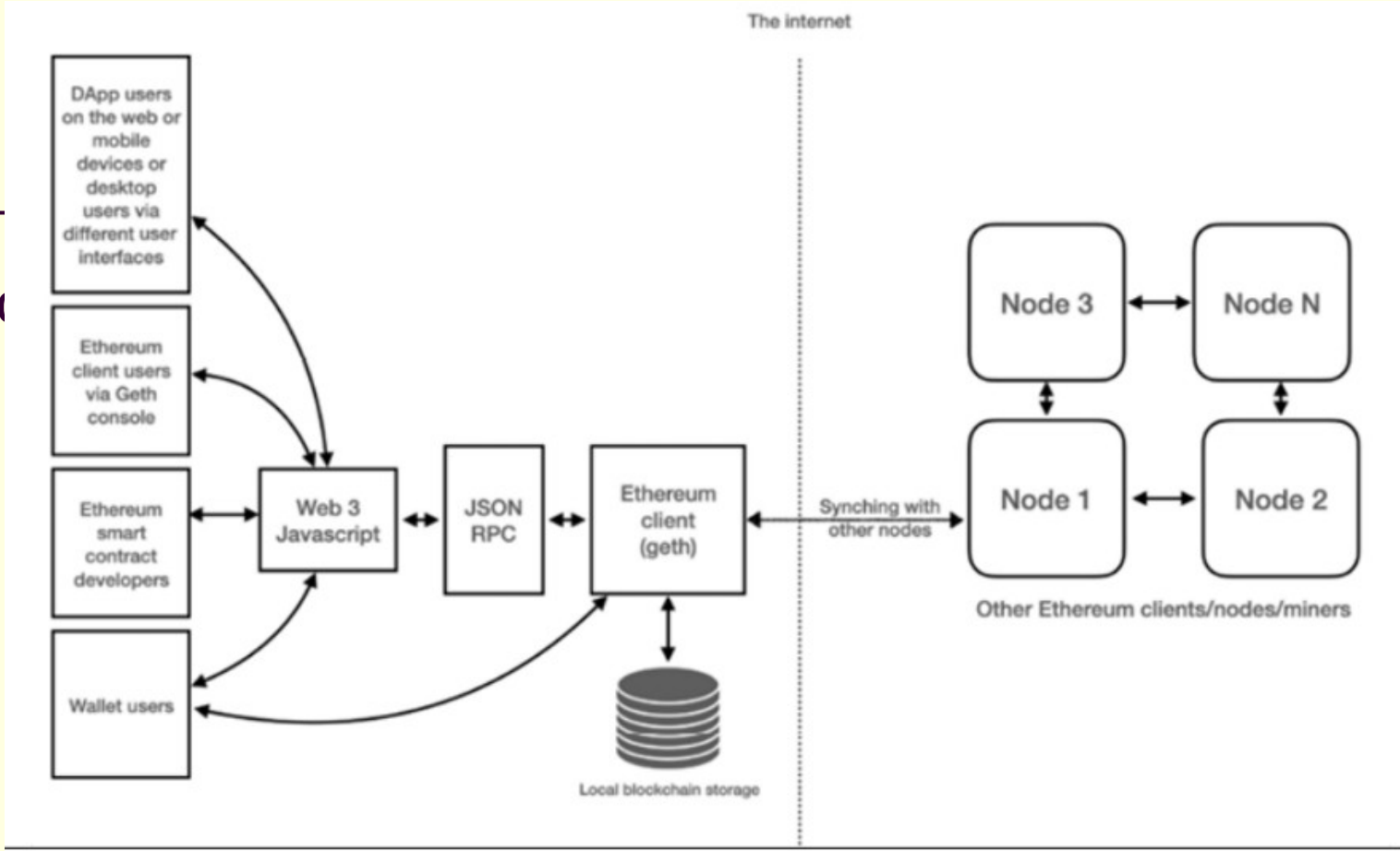


Figure 11.6: Ethereum high-level ecosystem

The Ethereum network

- elements present in the Ethereum blockchain is
  - Keys and addresses
  - Accounts
  - Transactions and messages
  - Ether cryptocurrency/tokens
  - The EVM
  - Smart contracts and native contracts

Keys and addresses

- to represent ownership and transfer ether
- keys are made up of pairs of private and public parts
- Private key is generated randomly and is kept secret, whereas a public key is derived from the private key
- Addresses are derived from public keys and are 20-byte codes used to identify accounts

Keys and addresses

The process of key generation and address derivation is as follows:

1. First, a private key is randomly chosen (a 256-bit positive integer) under the rules defined by the elliptic curve `secp256k1` specification (in the range [`1` , `secp256k1n - 1`]).
2. The public key is then derived from this private key using the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) recovery function. We will discuss this in the following *Transactions and messages* section, in the context of digital signatures.
3. An address is derived from the public key, specifically, from the rightmost 160 bits of the Keccak hash of the public key.

Accounts

- one of the main building blocks of the Ethereum blockchain
- defined by pairs of private and public keys
- used to interact with the blockchain via transactions
- A transaction is digitally signed by an account before submitting it to the network via a node
- Ethereum, being a transaction-driven state machine, the state is created or updated as a result of the interaction between accounts and transaction executions

Accounts

- All accounts have a state that, when combined together, represents the state of the Ethereum network
- With every new block, the state of the Ethereum network is updated
- Operations performed between and on the accounts represent state transitions
- The state transition is achieved using Ethereum state transition function

Accounts

- Working of Ethereum state transition function
    - Confirm the transaction validity by checking the syntax, signature validity, and nonce
    - The transaction fee is calculated, and the sending address is resolved using the signature. the sender's account balance is checked and subtracted accordingly, and the nonce is incremented. An error is returned if the account balance is insufficient

Accounts

- Working of Ethereum state transition function
    - Provide enough ETH (the gas price) to cover the cost of the transaction
    - This is charged per byte and is incrementally proportional to the size of the transaction
    - In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back except for the fee payment, which is paid to the miners

Accounts

- Working of Ethereum state transition function
    - the remainder (if any) of the fee is sent back to the sender as change and the fee is paid to the miners accordingly. At this point, the function returns the resulting state, which is also stored on the blockchain

Accounts

- Two types
  - Externally Owned Accounts (EOAs) - similar to accounts that are controlled by a private key in Bitcoin
  - Contract Accounts (CAs) - accounts that have code associated with them along with the private key

Properties of Accounts

EAOs	CAs
• have a state • associated with a human user, hence are also called user accounts • EOAs have an ether balance • capable of sending transactions • no associated code • controlled by private keys • EOAs cannot initiate a call message • Accounts contain a key-value store • EOAs can initiate transaction messages	• have a state • not intrinsically associated with any user or actor on the blockchain • CAs have an ether balance • They have associated code that is kept in memory/storage on the blockchain • have access to storage • can get triggered and execute code in response to a transaction or a message from other contracts • can maintain their permanent states and can call other contracts • cannot start transaction messages • can initiate a call message • contain a key-value store • addresses are generated when they are deployed

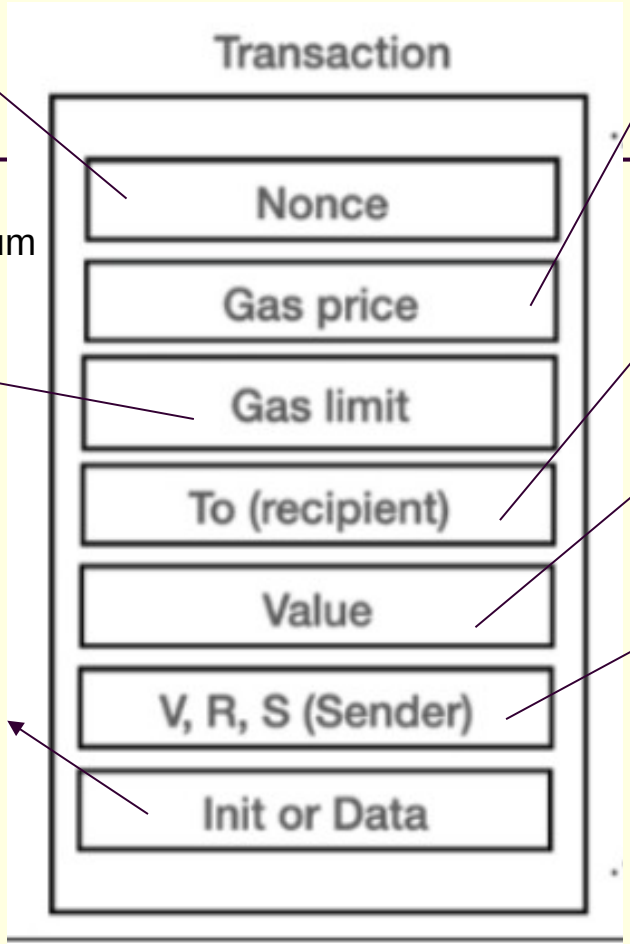
Transactions

- A transaction in Ethereum - digitally signed data packet using a private key that contains the instructions that, when completed, either result in a message call or contract creation
- two types based on the output they produce:
    - Message call transactions: simply produces a message call that is used to pass messages from one CA to another
    - Contract creation transactions: result in the creation of a new CA

Unique identifier incremented by one every time a transaction is sent by the sender

value that represents the maximum amount of gas that can be consumed to execute the transaction

Init field is used only in transactions that are intended to create contracts
If the transaction is a message call, then the Data field is used instead of init, and represents the input data of the message call



amount of Wei required to execute the transaction

20 byte value address

total number of Wei to be transferred to the recipient

digital signature (R, S) and some information that can be used to recover the public key (V)

Wei is the smallest denomination of ether

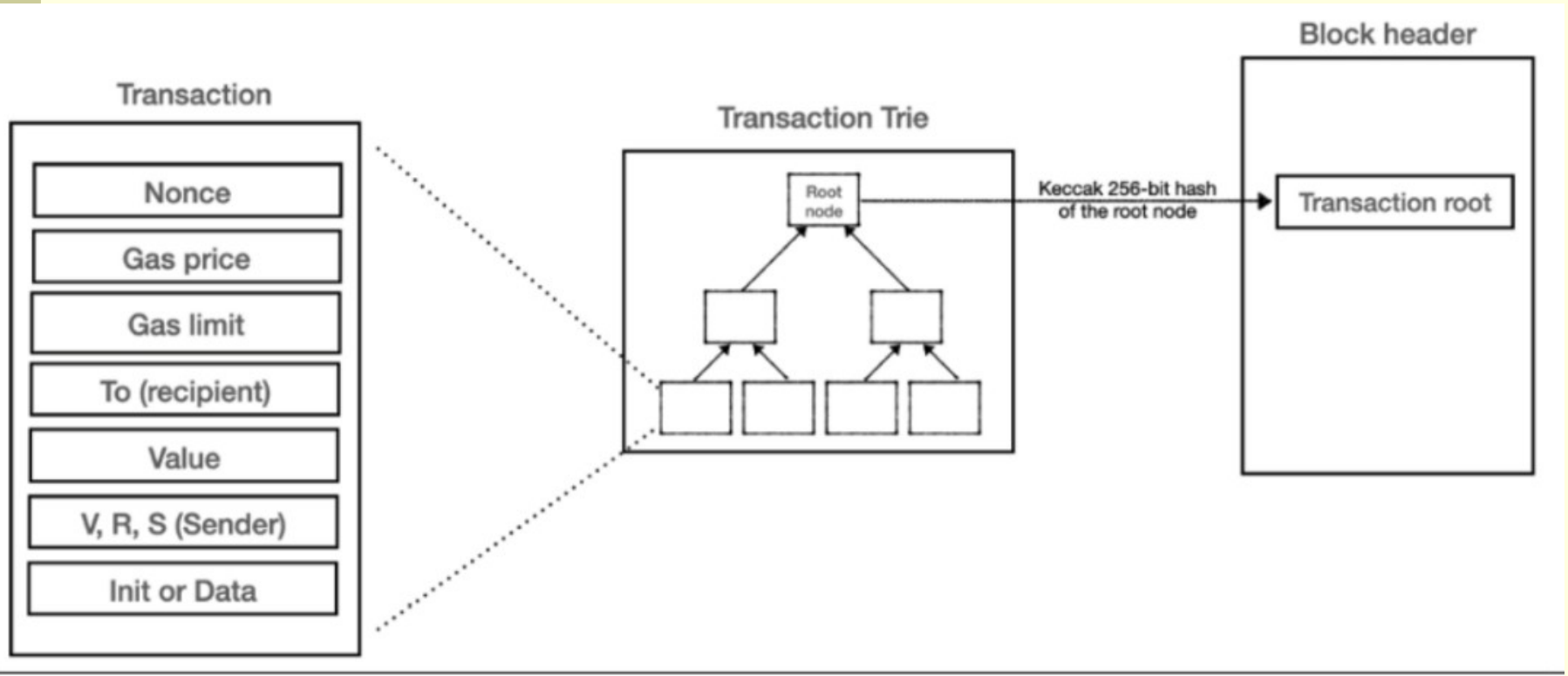


Figure 11.7: The relationship between the transaction, transaction trie, and block header

Messages

- a data packet passed between two accounts
- can either be sent via a smart contract (autonomous object) or from an external actor (an Externally Owned Account, or EOA) in the form of a transaction that has been digitally signed by the sender
- Contracts can send messages to other contracts
- Messages only exist in the execution environment and are never stored

Messages

- Messages are produced by the contracts, whereas transactions are produced by entities external to the Ethereum environment (EOAs)
- Messages are generated when the CALL or DELEGATECALL opcodes are executed by the contract running in the EVM

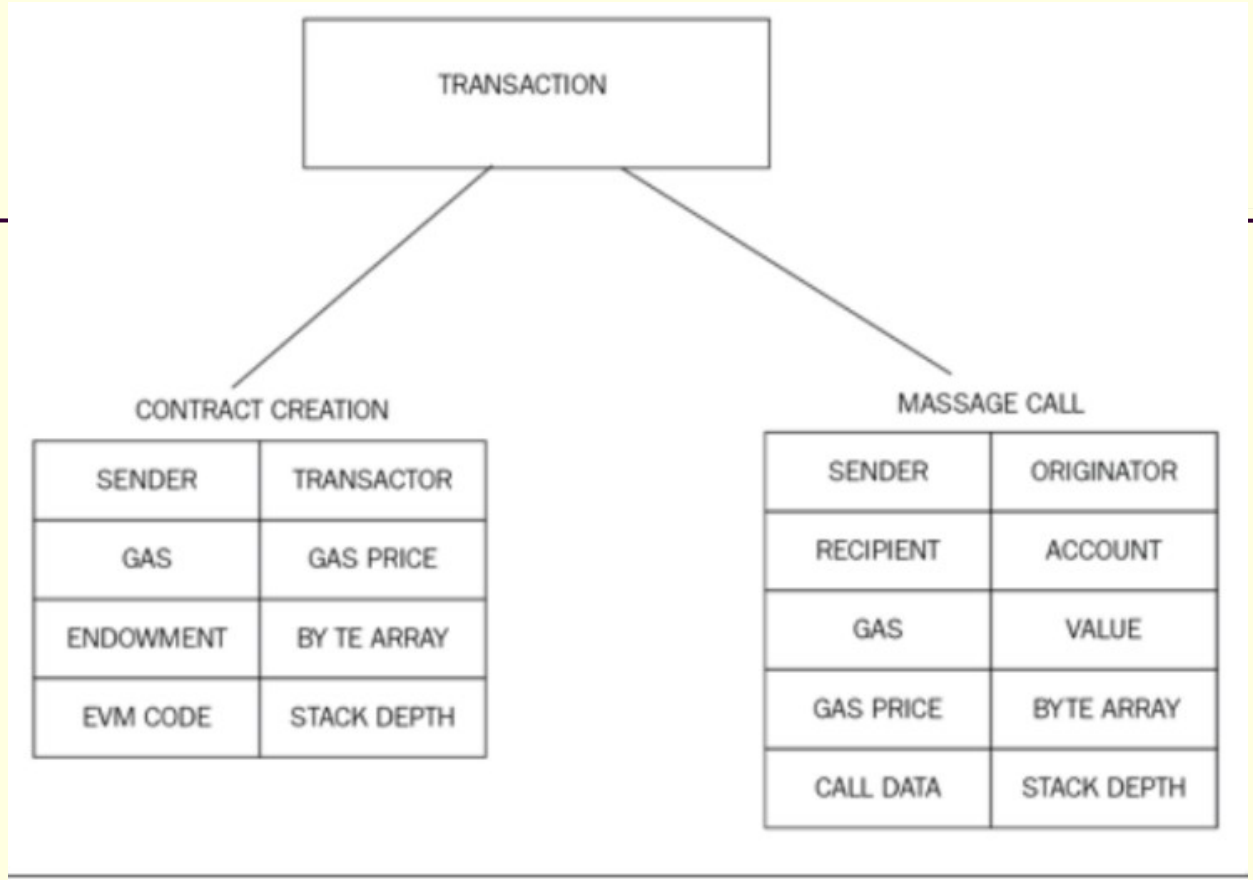


Figure 11.8: Types of transactions and the required parameters for execution

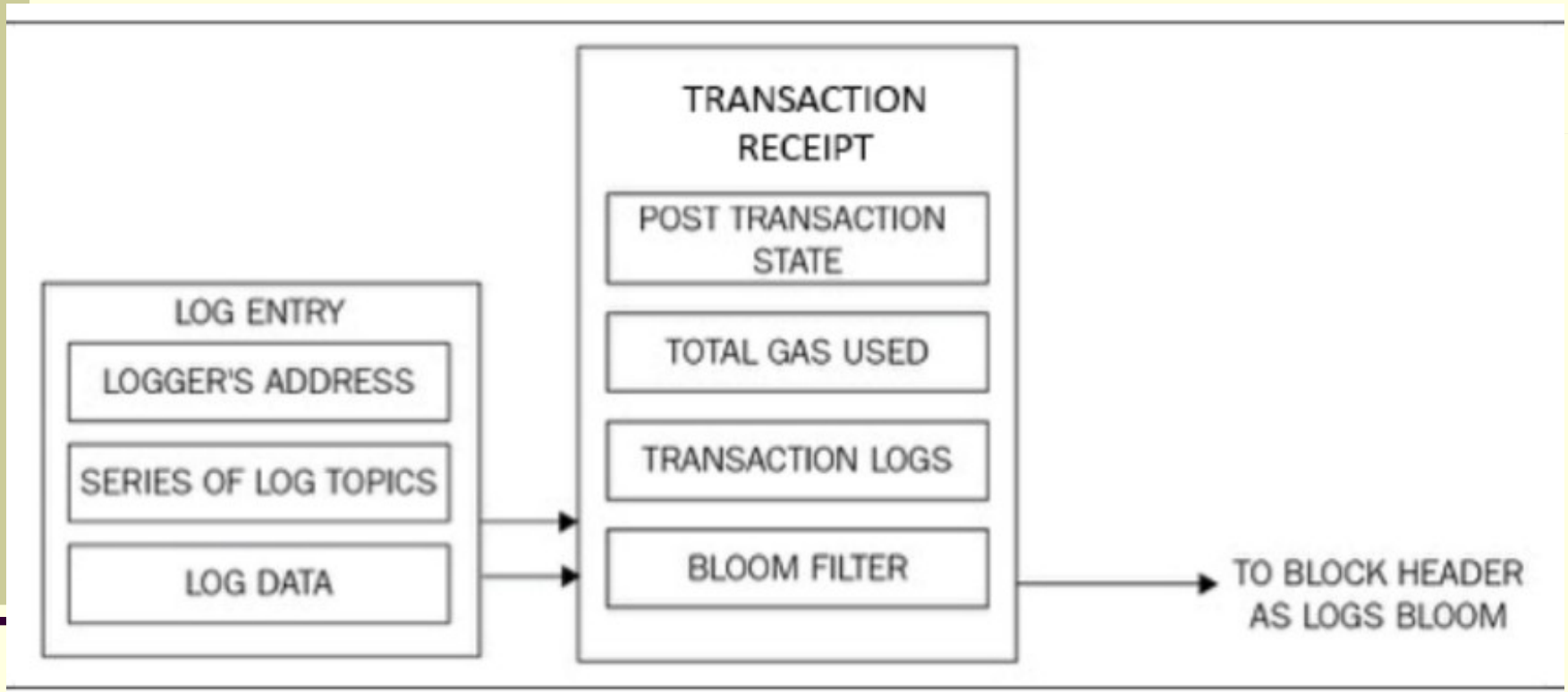


Figure 11.10: Transaction receipts and logs bloom

The Ethereum Virtual Machine (EVM)

- simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another
- word size of the EVM is set to 256-bit
- stack size is limited to 1,024 elements based LIFO
- Turing-complete machine but is limited by the amount of gas that is required to run any instruction

The Ethereum Virtual Machine (EVM)

- infinite loops that can result in denial-of-service attacks are not possible due to gas requirements
- Supports exception handling by immediately halt and return the error to the executing agent
- EVM is an entirely isolated and sandboxed runtime environment
- The code that runs on the EVM does not have access to any external resources such as a network or filesystem

The Ethereum Virtual Machine (EVM)

- This results in increased security, deterministic execution, and allows untrusted code (code that can be run by anyone) to be executed on Ethereum blockchain

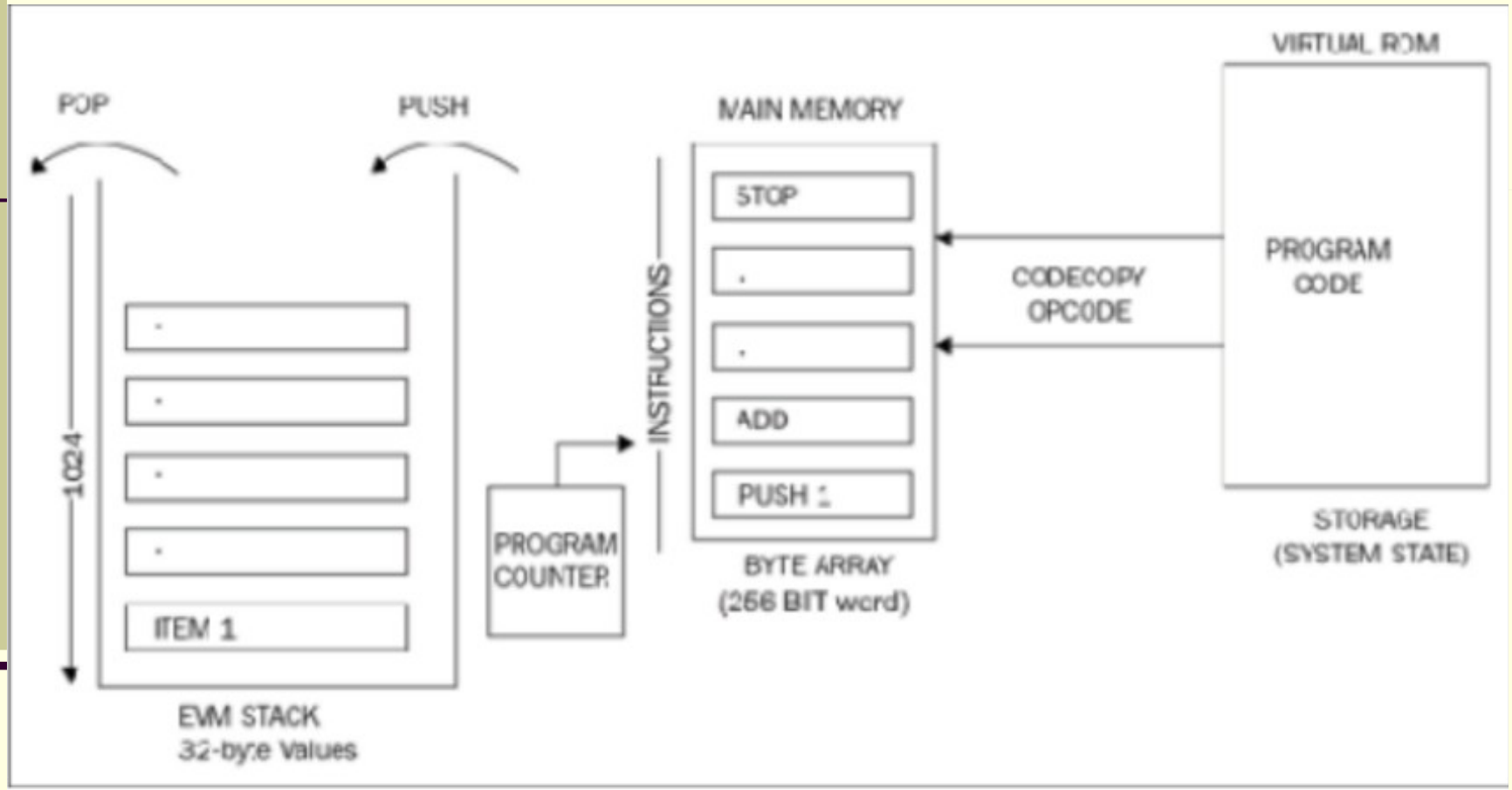
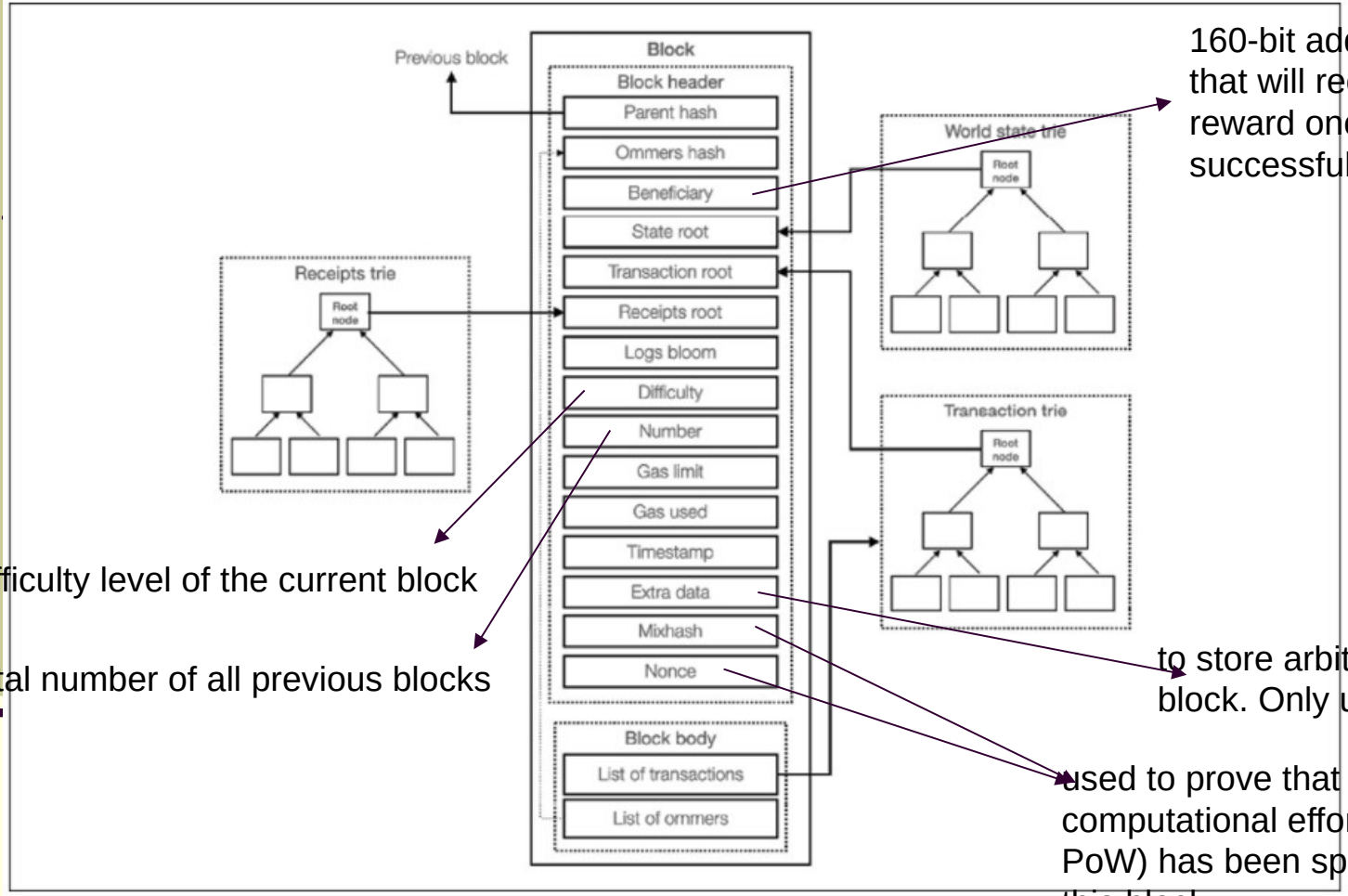


Figure 11.11: EVM operation

Blocks and blockchain

- main building structure of a blockchain
- Ethereum blocks consist of
    - The block header
    - The transactions list
    - The list of headers of ommers or uncles

An uncle block is a block that is the child of a parent but does not have any child block. Ommers or uncles are valid, but stale, blocks that are not part of the main chain but contribute to security of the chain. They also earn a reward for their participation but do not become part of the canonical truth.



160-bit address of the recipient that will receive the mining reward once the block is successfully mined.

difficulty level of the current block

total number of all previous blocks

to store arbitrary data related to the block. Only up to 32 bytes

used to prove that adequate computational effort (Proof of Work, or PoW) has been spent in order to create this block.

Figure 12.1: A detailed diagram of the block structure with a block header and relationship with tries

Block finalization

- Block finalization is a process that is run by miners to validate the contents of the block and apply rewards
- 4 steps
    - Ommers validation - checks whether the header is valid and whether the relationship between the uncle and the current block satisfies the maximum depth of six blocks and a block can contain a maximum of two uncles

Block finalization

- 4 steps
  - Transaction validation - checking whether the total gas used in the block is equal to the final gas consumption after the final transaction
  - Reward application - updating the beneficiary's account with a reward balance. Current block reward is 2 ether
  - State and nonce validation - compute a valid state and block nonce

Block difficulty mechanism

- Block difficulty is increased if the time between two blocks decreases, whereas it increases if the block time between two blocks decreases
- This is required to maintain a roughly consistent block generation time
- The difficulty adjustment algorithm in Ethereum's Homestead release is as follows

Block difficulty mechanism

- if the time difference between the generation of the parent block and the current block is less than 10 seconds, the difficulty goes up by parent_diff // 2048 * 1
- If the time difference is between 10 and 19 seconds, the difficulty level remains the same
- Finally, if the time difference is 20 seconds or more, the difficulty level decreases
- This decrease is proportional to the time difference from parent_diff // 2048* -1 to a maximum decrease of parent_diff // 2048 * -99

Block difficulty mechanism

$$block\_diff = parent\_diff + parent\_diff // 2048 *$$

$$max(1 - (block\_timestamp - parent\_timestamp) // 10, -99) +$$

$$int(2**((block.number // 100000) - 2))$$

Note that // is the integer division operator.

Block difficulty mechanism

In addition to timestamp difference-based difficulty adjustment, there is also another part (shown in the last line of the preceding algorithm) that increases the difficulty exponentially after every 100,000 blocks. This is the so-called **difficulty time bomb**, or **ice age**, introduced in the Ethereum network, which will make it very hard to mine on the Ethereum blockchain at some point in the future. This will encourage users to switch to **Proof of Stake** (**PoS**), since mining on the PoW chain will eventually become prohibitively difficult.

Gas

- Gas is required to be paid for every operation performed on the Ethereum blockchain
- mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM
- transaction fee is charged as an amount of Ether and is taken from the account balance of the transaction originator

Gas

- A fee is paid for transactions to be included by miners for mining
- If this fee is too low, the transaction may never be picked up
- the more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block
- if the transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an out-of-gas exception if the gas cost is not enough

Gas

- the transaction will fail but will still be made part of the block, and the transaction originator will not get any refund
- Transaction costs can be estimated using the following formula:

  Total cost = gasUsed * gasPrice
- gasUsed is the total gas that is supposed to be used by the transaction during the execution, and gasPrice is specified by the transaction originator as an incentive to the miners to include the transaction in the next block

Gas

- This is specified in ETH
- Each EVM opcode has a fee assigned to it
- It is an estimate because the gas used can be more or less than the value specified by the transaction originator originally

Solidity

- statically-typed (type checking at compile time) curly-braces object-oriented programming language designed for developing smart contracts that run on Ethereum
- domain-specific language of choice for programming contracts in Ethereum
- syntax is closer to both JavaScript and C
- also called a contract-oriented language

The layout of a Solidity source code file

- contracts are equivalent to the concept of classes
- three types of variables in Solidity: local variables, global variables, and state variables
- State variables have their values permanently stored in smart contract storage
- State variables are declared outside the body of a function, and they remain available throughout the contract depending on the accessibility assigned to them and as long as the contract persists

```
1  pragma solidity ^0.5.0; //specify the solidity compiler version
2  /*
3  this is a simple value checker contract that checks the value provided
4  and returns boolean value (true or false) based on the condition expression
5  evaluation
6  */
7  import "./mapping.sol"; //import a file
8  contract valuechecker {
9      uint price = 10;
10     //price variable declared and initialized with a value of 10
11     event valueEvent(bool returnValue);
12     function Matcher (uint8 x) public returns (bool) {
13         if (x >= price )
14         {
15             emit valueEvent(true);
16             return true;
17         }
18     }
19 }
```

Figure 14.13: Sample Solidity program as shown in the Remix IDE

The layout of a Solidity source code file

- Solidity has two categories of data types: value types and reference types
- Reference types store the address of the memory location where the value is stored
- Value types mainly include Booleans, integers, addresses, and literals

Data Types

- Boolean - true or false

Keyword	Types	Details
`int`	Signed integer	`int8` to `int256`, which means that keywords are available from `int8` up to `int256` in increments of 8, for example, `int8`, `int16`, and `int24`.
`uint`	Unsigned integer	`uint8`, `uint16`, ... to `uint256`, unsigned integer from 8 bits to 256 bits. Usage is dependent on how many bits are required to be stored in the variable.

Data Types

- Address -holds a 160-bit long (20 byte) value
    - has several members that can be used to interact with and query the contracts
    - Balance: returns the balance of the address in Wei
    - Send: used to send an amount of ether to an address and returns true or false depending on the result of the transaction

Data Types

- Address -holds a 160-bit long (20 byte) value
    - Call functions: call , callcode , and delegatecall functions
        - to interact with functions that do not have an ABI (Application Binary Interface)
    - Array value types (fixed-size and dynamically sized byte arrays): Fixed-size keywords range from bytes1 to bytes32 , whereas dynamically sized keywords include bytes and string

Data Types

- Address -holds a 160-bit long (20 byte) value
    - The bytes keyword is used for raw byte data, and string is used for strings encoded in UTF-8
    - An example of a static (fixed size) array is as follows:

```
bytes32[10] bankAccounts;
```

    - An example of a dynamically sized array is as follows:

```
bytes32[] trades;
```

    - Get the length of trades by using the following code:

```
trades.length;
```

Data Types

- Literals - to represent a fixed value

- **Integer literals**: These are a sequence of decimal numbers in the range of 0–9. An example is shown as follows:

```
uint8 x = 2;
```

- **String literals**: This type specifies a set of characters written with double or single quotes. An example is shown as follows:

```
'packt' "packt"
```

- **Hexadecimal literals**: These are prefixed with the keyword `hex` and specified within double or single quotation marks. An example is shown as follows:

```
(hex'AABBCC');
```

- **Enums**: This allows the creation of user-defined types. An example is shown as follows:

```
enum Order {Filled, Placed, Expired };
Order private ord;
ord=Order.Filled;
```

Reference types

- include arrays, structs, and mappings
- Arrays - contiguous set of elements of the same size and type
- Arrays have two members, named length and push
- Structs - used to group a set of dissimilar data types under a logical group

```
pragma solidity ^0.4.0;
contract TestStruct {
  struct Trade
  {
    uint tradeid;
    uint quantity;
    uint price;
    string trader;
  }
  //This struct can be initialized and used as below
  Trade tStruct = Trade({tradeid:123, quantity:1, price:1, trade
}
```

Reference types

- Mappings – used for a key to value mapping

```
mapping (address => uint) offers;
```

This example shows that `offers` is declared as a mapping. Another example makes this clearer:

```
mapping (string => uint) bids;
bids["packt"] = 10;
```

This is basically a dictionary or a hash table, where string values are mapped to integer values. The mapping named `bids` has the string `packt` mapped to value `10`.

Control Structures

- `if` : If `x` is equal to `0`, then assign value `0` to `y`, else assign `1` to `z` :

```
if (x == 0)
    y = 0;
else
    z = 1;
```

- `do` : Increment `x` while `z` is greater than `1` :

```
do{
    x++;
} (while z>1);
```

- `while` : Increment `z` while `x` is greater than `0` :

```
while(x > 0){
    z++;
}
```

Control Structures

- `for`, `break`, and `continue`: Perform some work until `x` is less than or equal to `10`. This `for` loop will run `10` times; if `z` is `5`, then break the `for` loop:

```
for(uint8 x=0; x<=10; x++)
{
    //perform some work
    z++
    if(z == 5) break;
}
```

It will continue the work in a similar vein, but when the condition is met, the loop will start again.

- `return`: `return` is used to stop the execution of a function and returns an optional value. For example:

```
return 0;
```

It will stop the execution and return a value of `0`.

Events

- used to log certain events in EVM logs
- useful when external interfaces are required to be notified of any change or event in the contract
- These logs are stored on the blockchain in transaction logs
- Logs cannot be accessed from the contracts but are used as a mechanism to notify change of state or the occurrence of an event (meeting a condition) in the contract

In a simple example here, the `valueEvent` event will return `true` if the `x` parameter passed to the function `Matcher` is equal to or greater than `10`:

```
pragma solidity >=0.6.0;
contract valueChecker
{
    uint8 price=10;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Inheritance

- is keyword is used to derive a contract from another contract
- derived contract has access to all non-private members of the parent contract

```
pragma solidity >=0.6.0;
contract valueChecker
{
    uint8 price = 20;
    event valueEvent(bool returnValue);

    function Matcher(uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
contract valueChecker2 is valueChecker
{
    function Matcher2() public view returns (uint)
    {
        return price+10;
    }
}
```

Libraries

- deployed only once at a specific address and their code is called via the CALLCODE or DELEGATECALL opcode of the EVM
- code reusability
- similar to contracts and act as base contracts to the calling contracts

```
library Addition
{
    function Add(uint x,uint y) returns (uint z)
    {
        return x + y;
    }
}
```

Library declaration

This library can then be called in the contract, as shown here. First, it needs to be imported and then it can be used anywhere in the code. A simple example is shown as follows:

```
import "Addition.sol"
function Addtwovalues() returns(uint)
{
    return Addition.Add(100,100);
}
```

There are a few limitations with libraries; for example, they cannot have state variables and cannot inherit or be inherited. Moreover, they cannot receive ether either; this is in contrast to contracts, which can receive ether.

Functions

```
pragma solidity >5.0.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

In the preceding code example, with `contract Test1`, we have defined a function called `addition1()`, which returns an unsigned integer after adding `2` to the value supplied via the variable `x`, initialized just before the function.

Functions

- Two types
  - Internal functions - used only within the context of the current contract
  - External functions - called via external function calls
- A function in Solidity can be marked as a constant
- Constant functions cannot change anything in the contract; they only return values when they are invoked and do not cost any gas