21B12CS317: Introduction to Blockchian Technologies

END TERM EXAMINATION SOLUTIONS KEY (EVEN 2023)

1. **[CO1: 3 Marks] Mr. Ramu wants to be safe regarding the payment made to the raw material supply committed by Mr. Scott. As a blockchain engineer, identify, describe, and justify in detail the best technique that can be implemented to achieve non-repudiation with respect to payment and supply commitment made by Mr. Scott.**

   [1 Marks] By applying digital signatures to the messages related to payments and supply commitments, Mr. Ramu can have strong cryptographic evidence of Mr. Scott's participation and commitment. The use of digital signatures helps establish non-repudiation by leveraging the unique properties of asymmetric cryptography and cryptographic hashing, ensuring the integrity, authenticity, and accountability of the communicated information.

   [2 Marks] Non-repudiation - The sender of a message should not deny in future that he has not sent the message. It can be achieved using Digital signature in which two phases available 1) signature creation 2) signature verification.

   At A's side
   E(msg, Pk_B)

   Let A and B agreed on SHA-256() function
   now produce message digest of msg -> SHA-256(msg) - MD_A_msg
   E(MD_A_msg, Sk_A) = SIG(MD_A_msg)
   Encryption of MD_A_msg using A's secrete key is said to be Signature creation. Now this signature is placed in the packet and sent to B.

   [Source_ID, Source_port,
   E(msg, Pk_B),
   MD_msg,
   SIG(MD_msg),
   Dest_ID, Dest_port]

   This packet now will be received by user B

   B needs to decrypt it

   D(E(msg, Pk_B), Sk_B) = msg

   Since both A and B agreed on SHA-256, B now will use the hash function to get the message digest

   D(SIG(MD_A_msg), Pk_A) = MD_A_msg

   SHA-256(msg) = MD_B_msg

   if(MD_A_msg = MD_B_msg) then message not tampered and sent by A else intigrity voilated and A is not the source.

   this process is said to be signature verification.

2. **[CO2: 3 Marks] Discuss the challenges that a blockchain network may encounter if a random node is chosen to mine a block, and how can double-spending be prevented when designing cryptocurrencies using blockchain technology?**

   [1.5 Marks] Several challenges can arise when a random node is chosen to mine a block in a blockchain network are

- **Mining Power Distribution: If a random node is selected to mine a block, there is a possibility that the node with the highest mining power may not be chosen. This can result in a less efficient use of computational resources and a slower overall network performance.**
- **Security Risks: Randomly selecting a node for block mining may introduce security risks if the chosen node is compromised or malicious. A malicious node could attempt to manipulate transactions, disrupt consensus, or engage in double-spending attacks.**
- **Network Scalability: Randomly selecting a node for mining may not consider the scalability of the network. Some nodes may have limited resources or slower processing capabilities, leading to bottlenecks and delays in block validation and propagation.**

[1.5 Marks] Methods to employ for preventing double-spending in cryptocurrencies:

- **Consensus Mechanisms: Consensus algorithms ensure that all nodes in the network agree on the validity and order of transactions. Proof of Work (PoW) and Proof of Stake (PoS) are commonly used consensus mechanisms. PoW requires miners to solve complex mathematical problems to validate blocks, while PoS relies on the stake (ownership) of cryptocurrency to select block validators. These mechanisms make it difficult for an attacker to control the majority of the network's mining power and perform double-spending attacks.**
- **Block Confirmation: Each block added to the blockchain should have a certain number of subsequent blocks mined on top of it to increase its level of confirmation. The more confirmations a block has, the more secure and irreversible the included transactions become. Waiting for multiple confirmations minimizes the risk of double-spending, as it becomes increasingly computationally expensive and time-consuming to rewrite a significant portion of the blockchain.**
- **Transaction Validation: Nodes in the network independently validate and verify each transaction to ensure that the sender has sufficient funds to make the payment. This validation process checks the digital signatures, transaction history, and other relevant data to confirm the legitimacy of the transaction.**
- **Network Synchronization: Nodes maintain a synchronized view of the blockchain to prevent conflicting transactions and ensure consistency. By propagating valid blocks and transactions throughout the network, nodes can quickly detect and reject any attempts at double-spending.**
- **Decentralization: Distributing the blockchain across a large number of nodes and incentivizing their participation in the consensus process makes it difficult for any individual or group to manipulate the network for double-spending purposes. Decentralization ensures that no single entity has control over the entire blockchain, enhancing the security and integrity of the cryptocurrency system.**

3. **[CO3: 3 Marks] Mr. Eswar & Co is a startup firm wants to enter into Bitcoin mining business. The firm planned to procure a medium level mining processor. In this connection, illustrate various mining economic models available and suggest suitable model for the firm in view of its financial condition.**

[2 Marks] Mining economic models

- **Each mining pool contains hundreds or thousands of miners connected through special pool coordination protocols.**
- **Let B is the block reward minus pool fee, P = 1/D is the probability of finding a block in a shared attempt and D is the Block difficulty**
- **Pay per share (PPS) model**
  - **Share of miners R = B * (1/P)**
- **Proportional Share model**
  - **Payments are made once the pool finds the block.**
  - **Share of miners R = B * n/N**
    - **n is the amount of miner share**
    - **N is the amount of all shares in the round.**
  - **miner earn share until the pool finds a block.**

- **Pay per last N shares (PPLNS) model: It is similar to proportional model, in this miners' reward is calculated on the basis of last N shares.**
- **Advantages of mining pools: 1) Small miners can participate. 2) Predictable mining.**

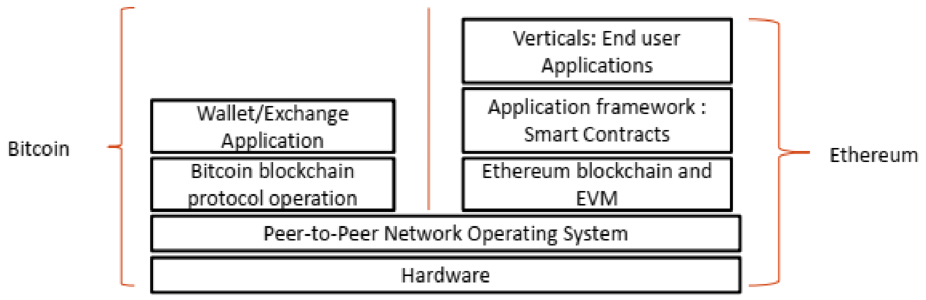
Disadvantages: 1) Leads to centralization, 2) Discourage miners for running complete mining.

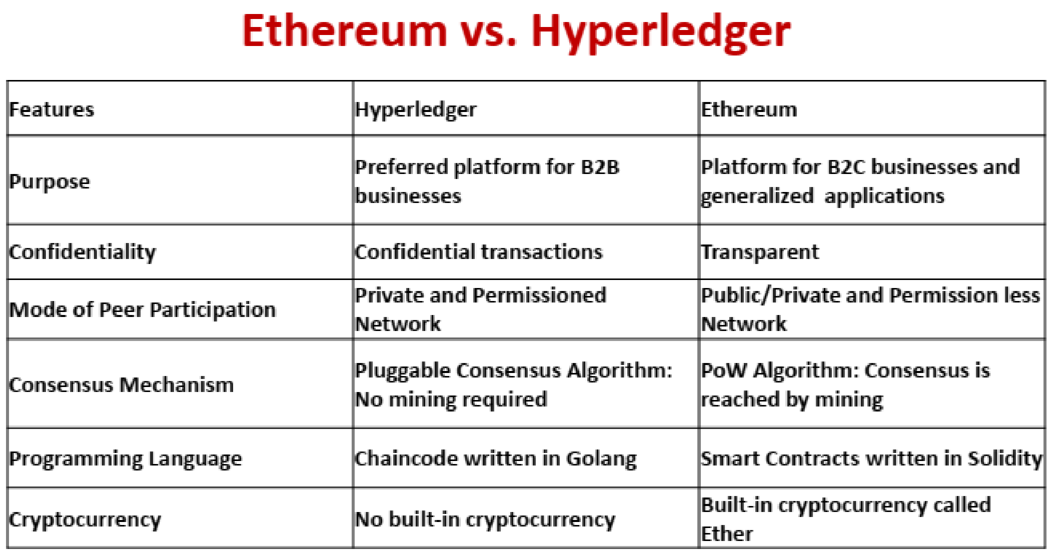
[1 Mark] Considering the financial condition of Mr. Eswar & Co, it is essential to choose a mining economic model that aligns with their budget and resources. Since the company is a start up, a suitable model would be the ROI model. Among the above models, Pay per share (PPS) can be considered as ROI model. This model allows the company to analyze the time it takes to recover their initial investment and start generating profits. By considering the upfront cost of the mining processor, ongoing operational costs, expected mining rewards, and current network difficulty, company can estimate their ROI period and make informed decisions about entering the Bitcoin mining business.

4. [CO4: 3 x 2 = 6 Marks] Write short note

a. Bitcoin vs. Ethereum vs. Hyperledger architecture.

The following figure shows the difference between Bitcoin and Ethereum

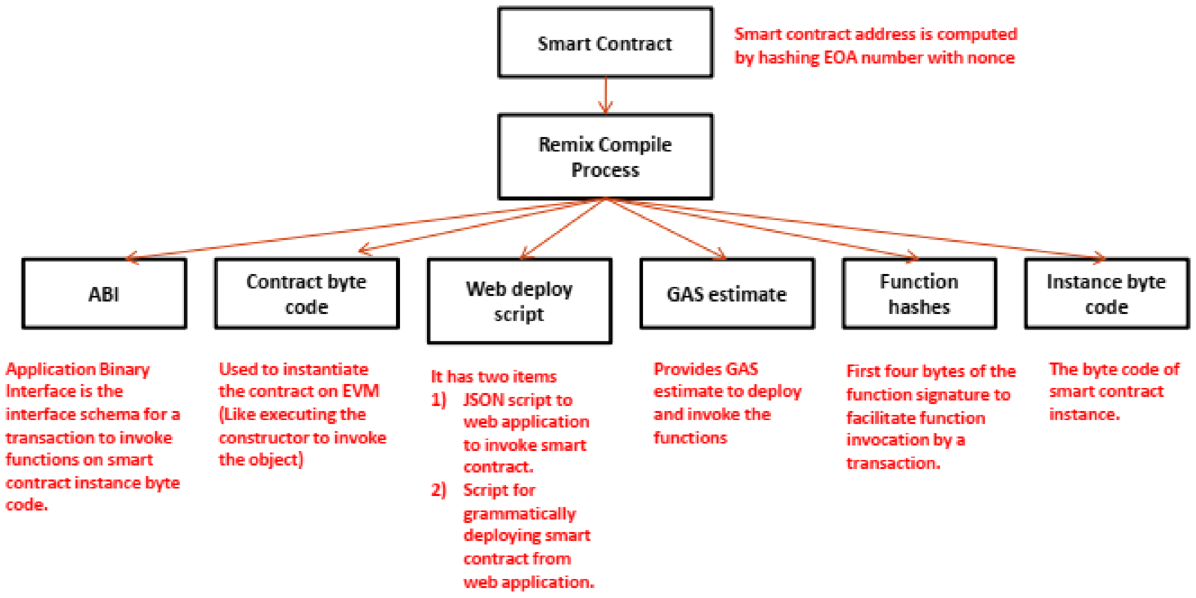

Bitcoin
Wallet/Exchange Application
Bitcoin blockchain protocol operation
Verticals: End user Applications
Application framework : Smart Contracts
Ethereum blockchain and EVM
Peer-to-Peer Network Operating System
Hardware
Ethereum



Ethereum vs. Hyperledger

Features	Hyperledger	Ethereum
Purpose	Preferred platform for B2B businesses	Platform for B2C businesses and generalized applications
Confidentiality	Confidential transactions	Transparent
Mode of Peer Participation	Private and Permissioned Network	Public/Private and Permission less Network
Consensus Mechanism	Pluggable Consensus Algorithm: No mining required	PoW Algorithm: Consensus is reached by mining
Programming Language	Chaincode written in Golang	Smart Contracts written in Solidity
Cryptocurrency	No built-in cryptocurrency	Built-in cryptocurrency called Ether

b. Artifacts generated by RemixIDE upon compiling smart contract.



Smart contract address is computed by hashing EOA number with nonce

Application Binary Interface is the interface schema for a transaction to invoke functions on smart contract instance byte code.

Used to instantiate the contract on EVM (Like executing the constructor to invoke the object)

It has two items
1) JSON script to web application to invoke smart contract.
2) Script for grammatically deploying smart contract from web application.

Provides GAS estimate to deploy and invoke the functions

First four bytes of the function signature to facilitate function invocation by a transaction.

The byte code of smart contract instance.

c. Mining and incentive models in Ethereum.

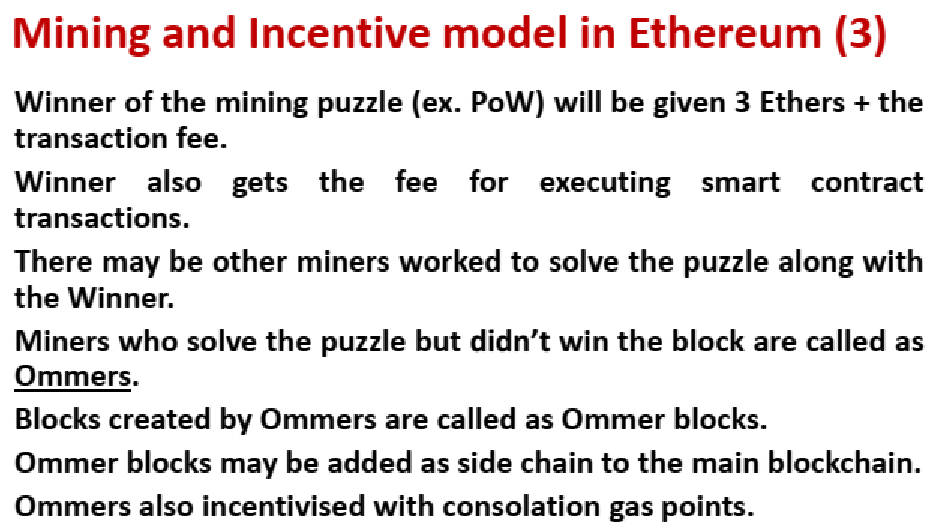
Mining and Incentive model in Ethereum (1)

- The condenses mechanism used in Ethereum is of memory based rather than processor based (PoW) used in Bitcoin.
- Every action in the Ethereum requires crypto fuel called Gas.
- Gas points are used to specify fee for Ether or ease of computation using standard values.
- Gas points are crypto-currency independent.
- Ethereum has defined specific Gas points for each type of operation.

Example: 1) Loading from memory	-	20 Gas points
2) Store in to memory	-	100 Gas points
3) Transaction base fee	-	21000 Gas points
4) Contract creation	-	53000 Gas points

Mining and Incentive model in Ethereum (2)

- Transaction can not be initiated if insufficient balance to pay mining fee in the account.
- The left over amount after paying the mining fee will be returned to originating account.
- In block, there are two Gas related items
  - 1) Gas limit – It is the amount of gas points available for a block to spent.
    - Example: For a block, GASLIMIT = 1.05 mUnits, Ether Tx fee = 21000, then block eligible for at most 70 plain Ether transactions.
    - Smart contracts required more gas points to execute.
  - 2) Gas spent – Actual amount spent at completion of block creation.



Mining and Incentive model in Ethereum (3)

Winner of the mining puzzle (ex. PoW) will be given 3 Ethers + the transaction fee.

Winner also gets the fee for executing smart contract transactions.

There may be other miners worked to solve the puzzle along with the Winner.

Miners who solve the puzzle but didn't win the block are called as Ommers.

Blocks created by Ommers are called as Ommer blocks.

Ommer blocks may be added as side chain to the main blockchain.

Ommers also incentivised with consolation gas points.

5. [CO4: 3 x 2 = 8 Marks] Answer the following questions w.r.t Solidity programming

a. Discuss the use of storage, memory, and stack keywords with an example.

Storage: The storage keyword is used to declare variables that persist between function calls and are stored permanently on the blockchain. Storage variables are expensive in terms of gas cost and should be used judiciously. The memory keyword is used to declare variables that are temporary and exist only during the execution of a function. Memory variables are cheaper in terms of gas cost compared to storage variables. Finally, the stack is a data structure used by the EVM (Ethereum Virtual Machine) to store local variables and intermediate values during the execution of a function. It is managed by the EVM and not explicitly declared by Solidity programmers.

Note on storage, memory, and stack:

1. Each account has a data area called storage, which is persistent between function calls and transactions. Storage is a key-value store that maps 256-bit words to 256-bit words.

2. The second data area is called memory, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. At the time of expansion, the cost in gas must be

paid. Memory is more costly the larger it grows (it scales quadratically).

3. The EVM is not a register machine but a stack machine, so all computations are performed on a data area called the stack. It has a maximum size of 1024 elements and contains words of 256 bits.

```
contract MyContract {
   uint256 myVariable; // Storage variable
   function setVariable(uint256 newValue) public {
      myVariable = newValue;
   }
   function getVariable() public view returns(uint256) {
      return myVariable;
   }
   function concatenateStrings(string memory a, string memory b) public pure returns(string memory) {
      return string(abi.encodePacked(a, b));
   }
}
```

b. Illustrate the difference between Abstract contract and interface with example.

An abstract contract is a contract that cannot be deployed on its own but serves as a base contract for other contracts to inherit from. It can contain both implemented and unimplemented functions. Derived contracts must provide implementations for all the unimplemented functions.

```
abstract contract MyAbstractContract {
   function myFunction() external virtual;

   function myImplementedFunction() external {
      // Implementation here
   }
}
contract MyContract is MyAbstractContract {
   function myFunction() external override {
      // Implementation here
   }
}
```

An interface is a contract that only contains function declarations without any implementation. It defines a set of function signatures that a contract must implement. Interfaces are typically used when interacting with external contracts or for providing interoperability between contracts.

```
interface MyInterface {
   function myFunction() external;
}
contract MyContract is MyInterface {
   function myFunction() external {
      // Implementation here
   }
}
```

c. **Develop a program to illustrate fallback function. And an event needs to be recorded every time the fallback function is called.**

```
pragma solidity ^0.8.0;

contract MyContract {
   event FallbackCalled(address caller, uint256 value);
   /*
         The fallback functions are a special type of function available only
         in Ethereum. A fallback function is invoked when no function name matches the
         called function.A fallback function does not have an identifier or function name. It
         is defined without a name. Since it cannot be called explicitly, it
         cannot accept any arguments or return any value.
         - writing a fall back function to by 1 ticket by default
      - a fallback function do not have any name
         - it must be declared as external (>0.5.0)
   */

   fallback() external payable {
      emit FallbackCalled(msg.sender, msg.value);
   }
}
```

d. **Let A, B, C, D, and E are smart contracts show the skeletal structure of various inheritances supported by solidity.**

```
contract A {
   function fooA() public pure virtual returns (string memory) {
      return "A";
```

```
    }
}

contract B is A {
    function fooB() public pure virtual returns (string memory) {
        return "B";
    }
}

contract C is A {
    function fooC() public pure virtual returns (string memory) {
        return "C";
    }
}

contract D is B, C {
    function fooD() public pure virtual returns (string memory) {
        return "D";
    }
}

contract E {
    function fooE() public pure virtual returns (string memory) {
        return "E";
    }
}

contract F is D, E {
    function fooF() public pure override(D, E) returns (string memory) {
        return string(abi.encodePacked(D.fooD(), " - ", E.fooE()));
    }
}
```

In this, contract A is the base contract inherited by contracts B and C. Contract D inherits from both B and C, allowing it to access functions from both parent contracts. Contract E is a separate contract unrelated to the others. Finally, contract F inherits from both D and E, and it overrides the fooF() function to concatenate the results of D.fooD() and E.fooE(). The virtual keyword is used to indicate that a function can be overridden, and the override keyword is used to explicitly specify the overridden function.

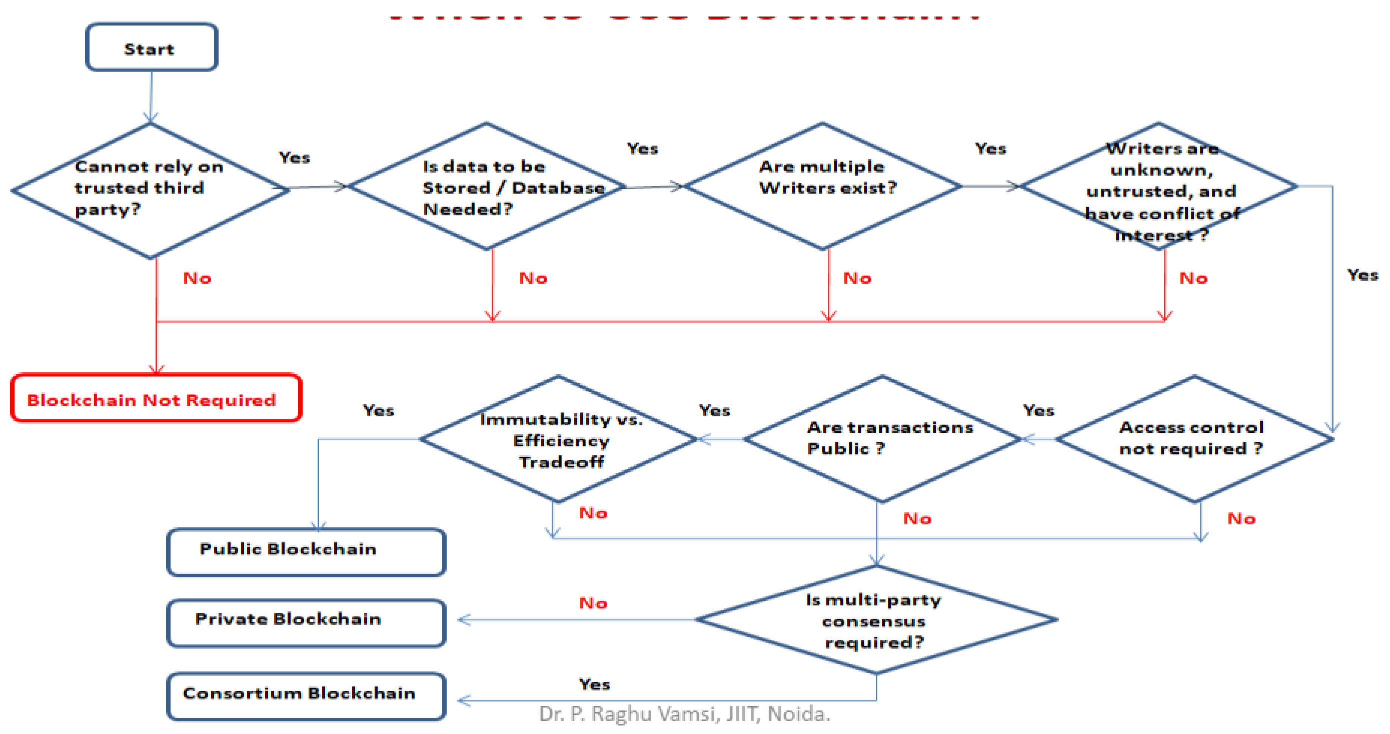
6. [CO5: 2 + 6 + 4 = 12 Marks] Consider the following case

ABC Insurance Ltd. is a company that provides various insurance services. The company board decided to introduce travel insurance for delay in fight journeys. The board decided to pay Rs. 10,000 or ticket price whichever is lower if the flight is delayed by at least 1 hour at the starting or destination air port or both. Anyone with age above 18 years can apply for this scheme. Currently, only company Finance manager can approve the new applications. If an approved customer wants to utilize this insurance then he/she needs to pay a premium amount of Rs.500 + (GST of 18%) at least 10 hours before the departure of the Airplane. At the end of the day, the collected GST will be credited to the government account. (Assume that 1 INR = 6192333890643383 Weis)

Answer the following questions

a. **Present with flow that what type of Blockchain is suggested in the above case.**

[1 Mark]



Start
Cannot rely on trusted third party?
Yes
Is data to be Stored / Database Needed?
Yes
Are multiple Writers exist?
Yes
Writers are unknown, untrusted, and have conflict of interest ?
No
No
No
No
Yes
Blockchain Not Required
Yes
Immutability vs. Efficiency Tradeoff
Yes
Are transactions Public ?
Yes
Access control not required ?
No
No
No
Public Blockchain
Private Blockchain
No
Is multi-party consensus required?
Consortium Blockchain
Yes
Dr. P. Raghu Vamsi, JIIT, Noida.

[1 Mark] From the case description, the application is between customer and service provider, also database is required, there will be multiple writers and they are unknown to each other. During the application running, access control is required and the transactions should be made private. Further, as GST is to be credited to government account, multiparty consensus is required for running the business. To this end, consortium blockchain is recommended for the application development.

b. **Develop Solidity smart contract to implement above case with the following modules with all required checks: 1) user registration, 2) application approval, 3) premium payment, 4) settlement, 5) GST payment to government.**

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

contract FlightInsurance {
    address public financeManager;
    uint256 public premiumAmount;
    uint256 public gstRate;
    uint256 public governmentAccount;

    struct Customer {
        bool isRegistered;
        bool isApproved;
        bool hasPaidPremium;
        uint256 ticketPrice;
    }

    mapping(address => Customer) public customers;

    event Registration(address indexed customer);
    event Approval(address indexed customer);
    event PremiumPaid(address indexed customer, uint256 amount);
    event Settlement(address indexed customer, uint256 amount);
    event GSTPaid(uint256 amount);

    constructor(uint256 _premiumAmount, uint256 _gstRate) {
        financeManager = msg.sender;
        premiumAmount = _premiumAmount;
        gstRate = _gstRate;
        governmentAccount = 0;
    }

    modifier onlyFinanceManager() {
        require(msg.sender == financeManager, "Only finance manager can call this function");
        _;
    }

    modifier onlyRegisteredCustomer() {
        require(customers[msg.sender].isRegistered, "Customer is not registered");
        _;
    }

    modifier onlyApprovedCustomer() {
        require(customers[msg.sender].isApproved, "Customer is not approved");
        _;
    }

    modifier onlyPaidPremium() {
        require(customers[msg.sender].hasPaidPremium, "Customer has not paid premium");
        _;
    }

    function registerCustomer() external {
        require(!customers[msg.sender].isRegistered, "Customer is already registered");
        customers[msg.sender].isRegistered = true;
        emit Registration(msg.sender);
    }

    function approveCustomer(address _customer) external onlyFinanceManager {
        require(customers[_customer].isRegistered, "Customer is not registered");
        require(!customers[_customer].isApproved, "Customer is already approved");
        customers[_customer].isApproved = true;
        emit Approval(_customer);
    }

    function payPremium() external payable onlyRegisteredCustomer onlyApprovedCustomer {
        require(!customers[msg.sender].hasPaidPremium, "Premium already paid");
        require(msg.value >= premiumAmount, "Insufficient premium amount");

        uint256 gstAmount = (premiumAmount * gstRate) / 100;
        governmentAccount += gstAmount;
        emit GSTPaid(gstAmount);

        customers[msg.sender].hasPaidPremium = true;
        customers[msg.sender].ticketPrice = msg.value;
        emit PremiumPaid(msg.sender, msg.value);
    }

    function settleInsurance() external onlyPaidPremium {
        uint256 delayHours = getRandomNumber(); // Assume a function to get the delay hours

        if (delayHours >= 1) {
            uint256 settlementAmount = customers[msg.sender].ticketPrice < 10000 ? customers[msg.sender].ticketPrice : 10000;
            payable(msg.sender).transfer(settlementAmount);
            emit Settlement(msg.sender, settlementAmount);
        }
    }

    function getRandomNumber() internal view returns (uint256) {
        // Function to generate random number for demonstration purposes
        return uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao, msg.sender))) % 24;
    }

      function withdrawGovernmentAccount() external onlyFinanceManager {
          payable(financeManager).transfer(governmentAccount);
          governmentAccount = 0;
      }
  }

```

c. Using web3js/web3py, present step by step procedure for linking the developed smart contract in (b) with the front end application to display approved customers public addresses.

Step 1: Setup

Install Web3.js and Web3.py libraries.
Set up your front-end application framework (e.g., HTML/CSS/JavaScript) and back-end server (e.g., Node.js for Web3.js or Python for Web3.py).

Step 2: Smart Contract Deployment

Deploy the Solidity smart contract to a blockchain network (e.g., Ethereum) using tools like Remix or Truffle. Note the deployed contract address.

Step 3: Connect to the Blockchain Network

In your front-end application (JavaScript), initialize Web3.js and connect to the blockchain network by specifying the provider (e.g., Metamask/Ganache):

Using javascript

var web3 = new Web3(Web3.givenProvider);

In your back-end application (Python), initialize Web3.py and connect to the blockchain network by specifying the provider (e.g., Infura):

python

```
from web3 import Web3
web3 = Web3(Web3.HTTPProvider('https://localhost:8545'))
```

Step 4: Load the Smart Contract

In your front-end application, load the deployed smart contract by providing the contract's ABI (Application Binary Interface) and address:

javascript

```
var contractABI = <ABI>;
var contractAddress = "<contract-address>";
var contract = new web3.eth.Contract(contractABI, contractAddress);
```

In your back-end application, load the smart contract by providing the contract's ABI and address:
python

```
contractABI = <ABI>
contractAddress = "<contract-address>"
contract = web3.eth.contract(address=contractAddress, abi=contractABI)
```

Step 5: Retrieve Approved Customers' Addresses

In your front-end application, call a function in the smart contract to retrieve the approved customers' public addresses:
using javascript

```
contract.methods.getApprovedCustomers().call()
  .then(function(result) {
    console.log(result); // Display the approved customers' addresses
  })
  .catch(function(error) {
    console.error(error);
  });
```

In your back-end application, call a function in the smart contract to retrieve the approved customers' public addresses:

using python

```
approvedCustomers = contract.functions.getApprovedCustomers().call()
print(approvedCustomers)  # Display the approved customers' addresses
```

Step 6: Display the Approved Customers' Addresses

In the front-end application, update the HTML elements to display the approved customers' addresses: Assuming you have an HTML element with id "approvedCustomersList" to display the addresses

```
var approvedCustomersList = document.getElementById("approvedCustomersList");
approvedCustomersList.innerHTML = ""; // Clear the previous content

for (var i = 0; i < result.length; i++) {
  var address = result[i];
  var listItem = document.createElement("li");
  listItem.textContent = address;
  approvedCustomersList.appendChild(listItem);
}
```

In the back-end application, pass the approved customers' addresses to the front-end application for display.

Another way of coding (not an elaborated one its only skeletal code, however, other constraints such as modifiers, conditions etc mentioned in the above code to be used in it)

```
pragma solidity ^0.8.0;

contract TravelInsurance {
    uint constant public WEI_IN_ONE_INR = 6192333890643383;
    uint constant public PREMIUM_AMOUNT = 500 * WEI_IN_ONE_INR;
    uint constant public GST_PERCENTAGE = 18;
    uint constant public MAX_INSURANCE_AMOUNT = 10000 * WEI_IN_ONE_INR;
    address public financeManager;

    mapping(address => bool) public customers;
    mapping(address => bool) public insuranceClaims;

    event ApplicationApproved(address customer);
    event InsuranceClaimed(address customer, uint amount);

    constructor() {
        financeManager = msg.sender;
    }

    function applyForInsurance() external {
        require(msg.sender.age() >= 18, "Must be 18 years or older to apply for insurance");
        require(!customers[msg.sender], "Customer has already applied for insurance");
        customers[msg.sender] = true;
        emit ApplicationApproved(msg.sender);
    }

    function approveApplication(address customer) external {
        require(msg.sender == financeManager, "Only finance manager can approve applications");
        customers[customer] = true;
        emit ApplicationApproved(customer);
    }

    function claimInsurance() external payable {
        require(customers[msg.sender], "Customer has not applied for insurance");
        require(!insuranceClaims[msg.sender], "Insurance has already been claimed");
        uint premiumWithGST = PREMIUM_AMOUNT + (PREMIUM_AMOUNT * GST_PERCENTAGE / 100);
        require(msg.value >= premiumWithGST, "Insufficient premium payment");
        require(block.timestamp < (msg.value - premiumWithGST + block.timestamp) - 10 hours, "Premium
payment made less than 10 hours before departure");
        uint insuranceAmount = MAX_INSURANCE_AMOUNT > msg.value - premiumWithGST ? msg.value -
premiumWithGST : MAX_INSURANCE_AMOUNT;
        insuranceClaims[msg.sender] = true;
        payable(msg.sender).transfer(insuranceAmount);
        uint gstAmount = premiumWithGST * GST_PERCENTAGE / 100;
        payable(address(this)).transfer(gstAmount);
        emit InsuranceClaimed(msg.sender, insuranceAmount);
    }
}
```