**BCT**

**UT 2 Question Bank**

**October 2023**

**Module 3**

Q. 1 Explain the types of smart contract.

Q. 2 Differentiate between dynamic arrays and fixed arrays.

Q. 3 Write a solidity code on the given problem statement.

Q. 4 Explain error handling in smart contracts.

**Module 4**

Q. 1 Explain Ethereum and components of Ethereum.

Q. 2 Differentiate between EOA and Contract accounts.

Q. 3 Compare Bitcoin and Ethereum.

Q. 4 Short note on test networks in Ethereum.

**Module 5**

Q. 1 Explain state machine replication?

Q. 2 Explain PAXOS consensus algorithm.

Q. 3 Explain Byzantine Fault Tolerant Consensus Protocol used in private blockchain.

Q. 4 Write short note on Hyperledger tools.

**Module 3**

1. **1 Explain the types of smart contract.**

Sure, I can explain the types of smart contracts. Smart contracts are self-executing contracts with the terms of the agreement directly written into code. They run on a blockchain and automatically enforce the rules and conditions of the contract. There are primarily two types of smart contracts:

Simple Smart Contracts:

These are basic smart contracts that involve a straightforward set of rules and conditions.

They are typically used for simple transactions, such as sending cryptocurrency from one wallet to another.

Simple smart contracts are easy to understand and execute because they involve only basic logic.

Complex Smart Contracts:

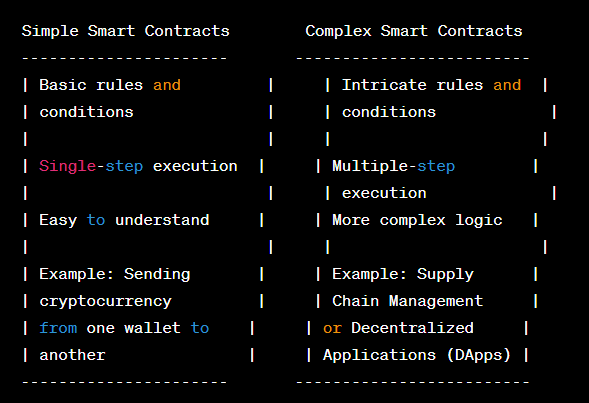
Complex smart contracts are more advanced and can involve intricate sets of rules and conditions.

They can be used for a wide range of applications, including decentralized applications (DApps), supply chain management, voting systems, and more.

Complex smart contracts often require multiple steps or conditions to be met before executing actions.

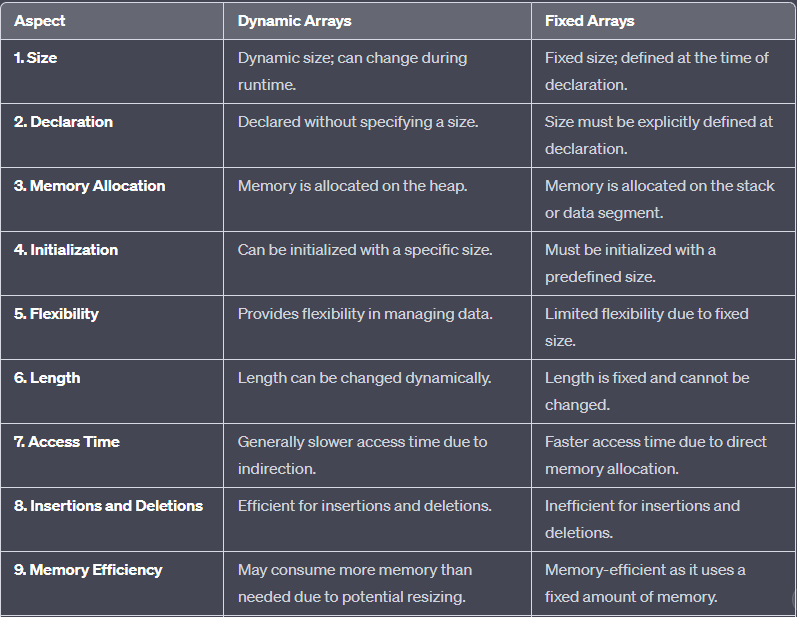
They can include functions like conditional statements, loops, and interactions with external data sources known as oracles.

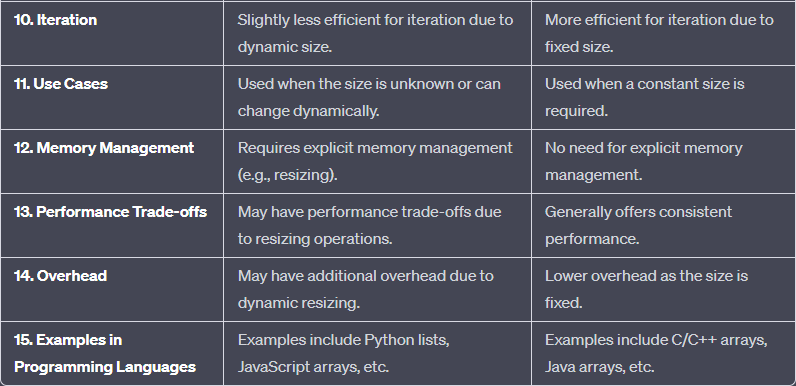
Here's a simple text-based diagram to illustrate the difference:



It's important to note that the distinction between simple and complex smart contracts can be somewhat subjective, as complexity depends on the specific use case and requirements of the contract. Smart contracts can vary in complexity from very basic to highly sophisticated based on the needs of the application they support.

1. **2 Differentiate between dynamic arrays and fixed arrays.**





**Q. 3 Write a solidity code on the given problem statement.**

Transfer Account:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract TransferAccount {

address public owner;

constructor() {

owner = msg.sender;

}

function transferTo(address payable recipient, uint256 amount) public payable {

require(msg.sender == owner, "Only the owner can transfer funds.");

require(address(this).balance >= amount, "Insufficient balance in the contract.");

recipient.transfer(amount);

}

}

2. Transfer Money Between Two Accounts:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract TransferMoney {

function transferTo(address payable recipient, uint256 amount) public payable {

require(address(this).balance >= amount, "Insufficient balance in the contract.");

recipient.transfer(amount);

}

}

1. Check Account Balance:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract CheckBalance {

function getContractBalance() public view returns (uint256) {

return address(this).balance;

}

}

**4 Explain error handling in smart contracts.**

Error handling in smart contracts is a crucial aspect of ensuring the security and robustness of your decentralized applications (DApps) on blockchain platforms like Ethereum. Proper error handling helps prevent unexpected behavior and vulnerabilities in your smart contracts. Here's an explanation of error handling techniques in smart contracts:

Require Statements:

The require statement is commonly used for error handling in smart contracts. It allows you to specify conditions that must be met for a function to proceed. If the condition isn't met, the function will revert, and any changes made up to that point will be rolled back.

Example:

function withdraw(uint256 amount) public {

require(amount <= balances[msg.sender], "Insufficient balance.");

// Perform the withdrawal if the condition is met.

}

Modifiers:

Modifiers are reusable code snippets that can be applied to multiple functions. They are often used for access control and error handling. Modifiers can include require statements to check conditions before executing a function.

Example:

modifier onlyOwner {

require(msg.sender == owner, "Only the owner can call this function.");

\_; // Continue executing the function if the condition is met.

}

function changeOwner(address newOwner) public onlyOwner {

owner = newOwner;

}

Error Messages:

When using require, it's essential to include clear and descriptive error messages. These messages help developers and users understand why a transaction failed.

Handling External Calls:

When interacting with external contracts or addresses, it's crucial to handle potential errors or failures gracefully. Using checks like success for low-level calls or using higher-level libraries can help manage errors.

Example:

(bool success, ) = address(recipient).call{value: amount}("");

require(success, "Transfer failed.");

Gas Considerations:

Error handling can consume gas, which is the fee required to execute transactions on the blockchain. Be mindful of gas costs when implementing error handling, especially in loops or complex logic.

Fallback and Receive Functions:

Solidity provides fallback and receive functions that can be used to handle Ether transfers to a contract when no suitable function is called. It's essential to implement these functions correctly to handle unexpected Ether transfers safely.

Events:

Emitting events in error-handling scenarios can provide transparency and make it easier to debug issues. You can log events with relevant information when certain conditions are not met.

Testing:

Thoroughly test your smart contracts using various scenarios, including edge cases and error conditions. Tools like Truffle and frameworks like Hardhat provide testing capabilities to simulate transactions and check for expected outcomes.

Upgradability Considerations:

If your contract is upgradable, take care to handle errors gracefully during upgrades to avoid issues that could arise from changes in contract logic.

Documentation:

Document the error-handling strategies and conditions in your smart contract code to make it easier for other developers to understand and maintain.

**Module 4**

**1 Explain Ethereum and components of Ethereum.**

Ethereum is a decentralized, open-source blockchain platform that enables the creation of decentralized applications (DApps) and the execution of smart contracts. It was proposed by Vitalik Buterin in late 2013 and development began in early 2014, with the network going live on July 30, 2015. Ethereum is a significant advancement beyond the basic capabilities of cryptocurrencies like Bitcoin, as it offers a more versatile and programmable blockchain platform.

Key components of Ethereum include:

Ethereum Blockchain:

The Ethereum blockchain is a distributed ledger that records all transactions across the network. It serves as a public, immutable database that is secured by a network of nodes (computers) running Ethereum software.

Ether (ETH):

Ether is the native cryptocurrency of the Ethereum platform. It serves as a means of value transfer within the network and is used to pay for transaction fees (gas) and as a store of value.

Smart Contracts:

Smart contracts are self-executing agreements with the terms of the contract directly written into code. They run on the Ethereum blockchain and automatically enforce the rules and conditions of an agreement without the need for intermediaries. Solidity is a popular programming language for writing Ethereum smart contracts.

Decentralized Applications (DApps):

DApps are applications that run on the Ethereum blockchain, utilizing smart contracts for their backend logic. These applications can be decentralized in various ways, allowing for greater transparency, security, and censorship resistance.

Ethereum Virtual Machine (EVM):

The EVM is a Turing-complete virtual machine that executes smart contract code. It ensures that the code is executed consistently across all nodes in the network, providing security and trust in the outcome.

Gas:

Gas is a measure of computational work required to execute operations on the Ethereum network. When users initiate transactions or smart contract interactions, they pay a fee in Ether called gas to compensate miners for processing their requests. The gas cost depends on the complexity of the operation and the network's congestion.

Miners and Mining:

Miners are participants in the Ethereum network who validate transactions and add them to the blockchain. They are incentivized with Ether rewards for their work. Mining in Ethereum involves solving complex mathematical puzzles through a process called Proof of Work (PoW). However, Ethereum is transitioning to a Proof of Stake (PoS) consensus mechanism with the Ethereum 2.0 upgrade.

Nodes:

Nodes are computers running Ethereum software that maintain a copy of the blockchain and participate in network validation. There are full nodes that store the entire blockchain and lightweight nodes that rely on full nodes for data.

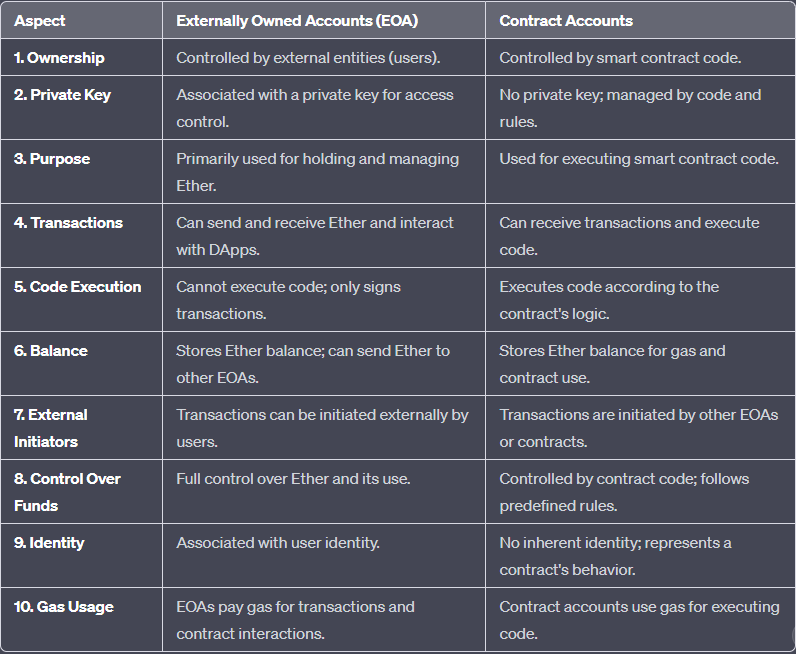
Wallets:

Ethereum wallets are software or hardware tools that allow users to manage their Ether and interact with the Ethereum network. They can be used to send and receive Ether, as well as interact with DApps and sign transactions.

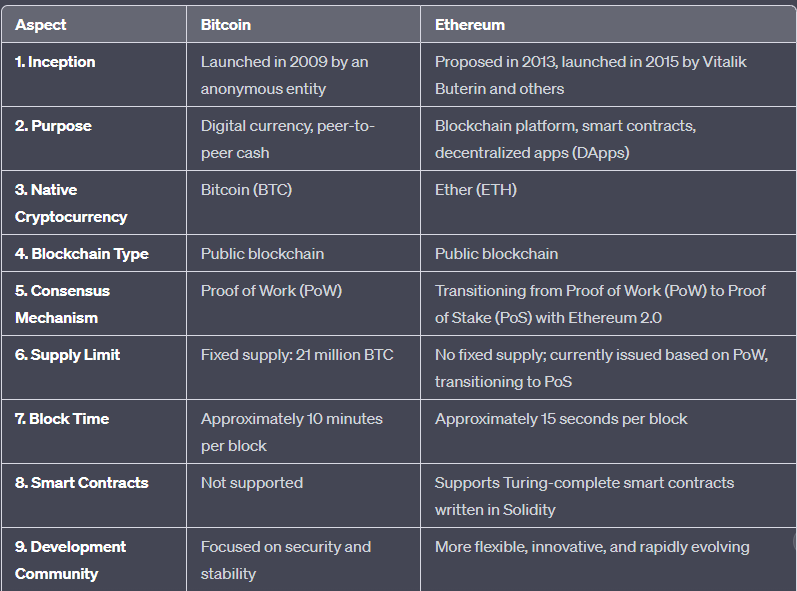
Consensus Mechanisms:

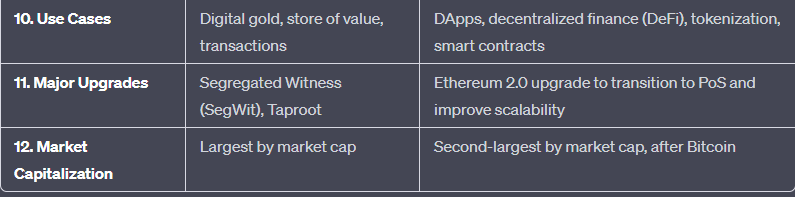
Ethereum initially used PoW as its consensus mechanism, but it is transitioning to PoS with Ethereum 2.0. PoS is more energy-efficient and aims to improve network scalability and security.

1. **2 Differentiate between EOA and Contract accounts.**



**Q. 3 Compare Bitcoin and Ethereum.**





**Q. 4 Short note on test networks in Ethereum.**

Test networks in Ethereum are parallel blockchains that mimic the behavior of the main Ethereum network (often referred to as the "mainnet") but with some key differences. These networks are essential for developers, testers, and anyone wanting to experiment with Ethereum without using real Ether (ETH) or risking real assets. Here's a short note on test networks in Ethereum:

1. Purpose:

Test networks exist to provide a safe and risk-free environment for testing and developing smart contracts, decentralized applications (DApps), and other Ethereum-related projects.

2. Test Ether (Testnet ETH):

Test networks have their own native tokens, often referred to as "Test Ether" or "Testnet ETH." These tokens have no real-world value and are freely available from testnet faucets or other sources.

3. Different Test Networks:

Ethereum has multiple test networks, including Ropsten, Rinkeby, Kovan, and Goerli, each with its own characteristics and purposes.

Ropsten, Rinkeby, and Kovan are public testnets, while Goerli is a cross-client testnet designed for interoperability between different Ethereum clients.

4. Testnet Faucets:

Testnet Ether can be obtained from testnet faucets, which are web services that distribute Testnet ETH to users for free. Users can request Testnet ETH by providing their testnet addresses.

5. Realistic Testing:

Test networks aim to closely mimic the behavior of the Ethereum mainnet, allowing developers to conduct realistic testing of their applications without risking real assets.

6. No Mining Rewards:

Test networks often disable mining rewards to discourage miners from participating and instead rely on a system where Test Ether is distributed for free.

7. Ethereum Client Compatibility:

Ethereum clients (software implementations of the Ethereum protocol) support these test networks, making it easy for developers to connect their Ethereum-based applications to test environments.

8. Development and Debugging:

Developers use test networks to deploy and test their smart contracts, identify and fix bugs, and ensure that their applications work as intended before deploying them on the mainnet.

9. Community Support:

The Ethereum community actively maintains and supports test networks to facilitate development and ensure the reliability of these environments.

10. Resettable:

- Test networks are occasionally reset, which means that their blockchains are cleared, and all Test Ether balances are reset to zero. This allows for a fresh start for testing.

In summary, test networks in Ethereum provide a vital playground for developers and enthusiasts to experiment, develop, and test their applications without incurring any real financial risk. They serve as a crucial resource for ensuring the security and functionality of Ethereum-based projects before they are deployed on the live Ethereum mainnet.

**Module 5**

**1 Explain state machine replication?**

State Machine Replication (SMR) is a distributed computing technique used to achieve fault tolerance and reliability in a distributed system. It is a fundamental concept in the field of distributed systems and is commonly employed in scenarios where high availability and consistency are crucial, such as in distributed databases, distributed file systems, and blockchain networks. Here's an explanation of state machine replication:

\*\*1. Definition:

State Machine Replication is a methodology for ensuring the consistency and fault tolerance of a distributed system by replicating the state of the system across multiple nodes and maintaining a consistent state across all replicas.

2. Key Components:

State Machine: The core of the system, which defines its behavior and state transitions. It takes input commands and transforms the system's state accordingly.

Replicas: Multiple copies or instances of the state machine that execute the same sequence of commands in the same order.

Client Requests: Requests or commands from clients that are submitted to the system for processing.

Replication Protocol: A protocol that ensures that all replicas execute the same commands in the same order and reach a consistent state.

3. Operation:

Clients send requests or commands to the replicas. These commands are processed by each replica's state machine.

The replicas execute the commands in the same order, ensuring that they transition through the same states.

The results of the commands are sent back to the clients, who observe the same behavior as if they interacted with a single, reliable system.

4. Fault Tolerance:

SMR provides fault tolerance because even if some replicas fail or become unreliable, as long as a quorum (a majority) of replicas are functioning correctly, the system can continue to operate reliably.

In the event of a replica failure, the system can elect a new leader or adjust the quorum to ensure continued operation.

5. Use Cases:

SMR is used in distributed systems that require high availability, consistency, and fault tolerance. Examples include databases, file systems, and distributed consensus algorithms in blockchain networks.

6. Consistency Guarantees:

SMR provides strong consistency guarantees, ensuring that all replicas arrive at the same state in the same order, as long as a quorum of replicas is operational.

7. Challenges:

Achieving high performance while maintaining strong consistency can be challenging due to the need for coordination among replicas.

Handling network partitions and latency requires careful design and may impact system performance.

8. Variants:

Various SMR variants exist, such as Primary-Backup Replication, Multi-Primary Replication, and Byzantine Fault Tolerant (BFT) Replication, each with different trade-offs and use cases.

In summary, State Machine Replication is a technique used to ensure the reliability and fault tolerance of distributed systems by replicating the state machine and executing commands in a consistent and coordinated manner across multiple replicas. It provides strong consistency guarantees and is essential for applications that require high availability and reliability.

**2 Explain PAXOS consensus algorithm.**

The Paxos consensus algorithm is a widely used and influential algorithm for achieving distributed consensus in a network of unreliable or faulty processes. It was introduced by Leslie Lamport in 1998 and is considered one of the fundamental algorithms in the field of distributed systems. Paxos is primarily used in situations where multiple nodes need to agree on a single value or decision, such as in distributed databases, replication systems, and fault-tolerant systems. Here's an explanation of the Paxos consensus algorithm:

1. Basic Idea:

The Paxos algorithm is designed to allow a group of distributed nodes to agree on a single value, even in the presence of network delays, node failures, and message loss. It ensures that once consensus is reached, all nodes agree on the chosen value.

2. Key Terminology:

Proposers: Nodes that initiate a proposal for a value.

Acceptors: Nodes that accept or reject proposals from proposers.

Learners: Nodes that eventually learn and store the agreed-upon value.

Ballot Number: A unique identifier for each proposal, typically containing a round number and the identifier of the proposer.

3. Phases of Paxos:

Phase 1 - Prepare: A proposer sends a "prepare" message to the acceptors with a ballot number. Acceptors respond with a "promise" to not accept any proposal with a lower ballot number.

Phase 2 - Accept: If a proposer receives promises from a majority of acceptors, it can send an "accept" message with its value. Acceptors respond by accepting the value if they haven't promised to accept a higher-numbered proposal.

4. Multiple Proposers:

Paxos can handle multiple proposers simultaneously trying to make decisions. A leader election process can be used to designate one proposer as the leader, which reduces contention.

5. Fault Tolerance:

Paxos is designed to tolerate failures, including the crash of proposers, acceptors, or network partitions. It ensures that consensus can still be reached as long as a majority of acceptors are functioning.

6. Liveness and Safety:

Paxos guarantees safety by ensuring that if a value is chosen, all nodes choose the same value. It also provides liveness by eventually allowing a value to be chosen as long as there are no network partitions lasting indefinitely.

7. Complexity and Variants:

The original Paxos algorithm is known for its complexity and can be challenging to implement correctly. Simplified variants like Multi-Paxos and Fast Paxos exist to improve efficiency and ease of implementation.

8. Real-World Use:

Paxos has influenced many real-world systems and distributed databases. Variations of Paxos are used in systems like Apache ZooKeeper and the Raft consensus algorithm.

In summary, the Paxos consensus algorithm is a foundational algorithm in distributed systems that allows a group of nodes to agree on a single value even in the presence of failures and network delays. It achieves safety and liveness guarantees, making it a key component in building fault-tolerant distributed systems.

**3 Explain Byzantine Fault Tolerant Consensus Protocol used in private blockchain.**

Byzantine Fault Tolerant (BFT) consensus protocols are designed to provide a high level of fault tolerance and security in distributed systems, including private blockchains. These protocols are especially valuable in scenarios where participants may be untrusted or where the network is susceptible to malicious actors or failures. Here, I'll explain the concept of Byzantine Fault Tolerant Consensus and its application in private blockchains:

1. Byzantine Fault Tolerance (BFT):

Byzantine Fault Tolerance is a property of a distributed system that allows it to continue functioning correctly and reach consensus even in the presence of a certain number of faulty or malicious nodes (Byzantine nodes).

The term originates from the "Byzantine Generals' Problem," a theoretical scenario where generals commanding an army must coordinate their attack plans, even when some of them may be traitors spreading misinformation.

2. BFT Consensus Protocols:

BFT consensus protocols aim to ensure that all nodes in a distributed network reach a common and agreed-upon state, even if some nodes are Byzantine (malicious) or experience arbitrary failures.

There are various BFT consensus algorithms, including Practical Byzantine Fault Tolerance (PBFT), HoneyBadgerBFT, and Tendermint, each with its own approach to achieving consensus.

3. Key Features of BFT Protocols:

Redundant Replicas: BFT protocols involve multiple replicas (nodes) that replicate the same data and execute the same set of transactions. This redundancy helps ensure consistency.

Voting and Agreement: Nodes participate in a voting process, where they exchange messages to propose, agree on, and commit to a common decision, such as the order of transactions or the state of the blockchain.

Quorum and Thresholds: BFT protocols often require a "quorum" of nodes to reach consensus, meaning a certain threshold or majority of nodes must agree before a decision is considered final.

Asynchronous Communication: BFT protocols assume that messages between nodes may experience unpredictable delays or be maliciously delayed, ensuring robustness against such scenarios.

4. Application in Private Blockchains:

Private blockchains are typically used within a closed network of known and trusted participants, such as organizations or consortiums.

BFT consensus protocols are preferred in private blockchains for their strong security guarantees, resilience against malicious nodes, and quick finality of transactions.

In a private blockchain, nodes are often pre-selected, and their behavior is more predictable than in public blockchains, making BFT more practical.

5. Trade-offs:

BFT consensus algorithms are known for their reliability and security but may sacrifice some degree of scalability compared to Proof of Work (PoW) or Proof of Stake (PoS) protocols.

The complexity and resource requirements of BFT algorithms can also be higher, which may be acceptable in private blockchains with fewer participants.

In summary, Byzantine Fault Tolerant (BFT) consensus protocols are crucial for ensuring the security and reliability of transactions in private blockchains. They are particularly valuable when dealing with known participants, where trust may be limited, and strong fault tolerance is required. BFT protocols enable these networks to reach consensus even in the presence of malicious actors or faulty nodes, making them a preferred choice for many enterprise and consortium blockchain applications.

**Q. 4 Write short note on Hyperledger tools.**

Hyperledger is an open-source project under the Linux Foundation that aims to advance the development of blockchain and distributed ledger technologies for enterprise use. It provides a range of tools and frameworks designed to support various aspects of blockchain development and deployment in business settings. Here's a short note on some of the key Hyperledger tools:

Hyperledger Fabric:

Hyperledger Fabric is one of the most prominent Hyperledger frameworks. It's a permissioned blockchain platform designed for enterprise use cases.

Key Features: Modular architecture, support for smart contracts (chaincode), private channels, and rich permissioning and identity management.

Hyperledger Sawtooth:

Hyperledger Sawtooth is a flexible and modular blockchain framework. It emphasizes simplicity and offers support for both permissioned and permissionless blockchains.

Key Features: Transaction family support, pluggable consensus mechanisms, and a unique addressing scheme.

Hyperledger Indy:

Hyperledger Indy is focused on providing digital identities and verifiable credentials. It's particularly useful in self-sovereign identity solutions.

Key Features: Identity wallets, decentralized identifiers (DIDs), and support for privacy-preserving identity solutions.

Hyperledger Besu:

Hyperledger Besu is an Ethereum-compatible client that can be used to build private or public Ethereum-based blockchains.

Key Features: Compatibility with Ethereum, support for enterprise use cases, and various consensus mechanisms.

Hyperledger Burrow:

Hyperledger Burrow is a permissioned Ethereum smart contract engine. It enables the execution of Ethereum smart contracts in a private network.

Key Features: Compatibility with Ethereum, support for EVM (Ethereum Virtual Machine), and permissioning features.

Hyperledger Iroha:

Hyperledger Iroha is designed for simplifying the development of applications that require a blockchain for identity and asset management.

Key Features: Simplified smart contract model, focus on mobile applications, and straightforward client libraries.

Hyperledger Cactus:

Hyperledger Cactus is a framework for integrating multiple blockchain platforms. It's designed to enable interoperability between different blockchains.

Key Features: Extensible architecture, support for various consensus algorithms, and a plugin-based approach for blockchain integration.

Hyperledger Transact:

Hyperledger Transact provides a set of libraries and utilities for building distributed ledger systems. It can be used with different Hyperledger blockchain platforms.

Key Features: Simplifies smart contract development, supports multiple languages, and provides consistency across platforms.

Hyperledger Quilt:

Hyperledger Quilt focuses on enabling interoperability between different blockchain networks and traditional financial systems.

Key Features: Implements the Interledger Protocol (ILP), supports payment channels, and facilitates cross-border transactions.

These Hyperledger tools and frameworks collectively offer a diverse set of resources for developing, deploying, and managing blockchain solutions in enterprise environments. Organizations can choose the most suitable tool or framework based on their specific use case and requirements, making Hyperledger a valuable resource for enterprise blockchain development.