Blockchain-Enabled Digital Twin Creation for Vehicles with Secure VIN Authentication and IoT Telemetry Integration

Dias Katrenov
dias.katrenov@studio.unibo.it

Mary Anne Selirio
maryanne.selirio@studio.unibo.it

November 2025

Disclaimer

During the preparation of this work, the author(s) used AI tools to assist with research organization and technical documentation. After using these tools, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

1 Introduction

The Blockchain-Enabled Digital Twin creation for Vehicles with secure vehicle identification number (VIN) authentication and IoT Telemetry Integration is a backend and dashboard system integrated with a Hyperledger Fabric blockchain. It allows secure registration, verification, and management of digital identities for vehicles with telemetry data collection from IoT devices. The system ensures Byzantine Fault Tolerance, Raft Consensus Protocol, Failure Detection and Recovery, Distributed Data Replication and Network Partition Tolerance. The project demonstrates the core principles of distributed systems through the integration of IoT sensor data with distributed blockchain storage, emphasizing data integrity, consensus protocols, and fault-tolerant replication.

2 Concept

This project presents a distributed blockchain system for secure vehicle service record management in the automotive domain. The system focuses on demonstrating core distributed systems principles through a VIN-based digital twin implementation that ensures data immutability, decentralized trust, and fault-tolerant consensus mechanisms. The integration with ESP32-based IoT telematics devices provides temperature and mileage data, demonstrating how distributed systems can provide Byzantine fault-tolerant storage for IoT data streams.

2.1 Product Type

The delivered solution consists of:

- **Permissioned Blockchain Network** using Hyperledger Fabric for decentralized vehicle record storage
- **Smart Contract System** automating vehicle onboarding and telemetry data validation
- **REST API Backend** (Node.js/Express) providing integration interfaces for IoT devices and external systems
- **Web Dashboard** demonstrating blockchain operations, telemetry visualization, and system monitoring capabilities
- **IoT Integration Layer** connecting ESP32 telematics devices via MQTT protocol with cryptographic authentication
- **Command-line Tools** for network administration and testing scenarios

2.2 Vehicle Digital Twin Focus

The system creates VIN-based digital twins for vehicles, addressing specific automotive industry challenges:

2.2.1 Vehicle Identity Management

- Unique, blockchain-anchored digital identity for each vehicle based on VIN
- Immutable ownership and service history tracking
- Cross-dealership service record verification capabilities
- Prevention of VIN duplication and fraudulent records
- Cryptographic authentication for IoT devices using VIN-derived Elliptic Curve Digital Signature Algorithm (ECDSA) keys

2.2.2 IoT Telemetry Data Management

- Automated telemetry logging from ESP32-based telematics devices
- Temperature monitoring using DS18B20 sensors
- Simulated mileage accumulation for demonstration purposes
- VIN-based cryptographic signatures for data authentication
- Byzantine fault tolerance ensuring IoT data integrity
- Distributed replication of telemetry across all blockchain peer nodes
- MQTT protocol for asynchronous IoT data transmission
- Integration with Embedded Systems & IoT course project hardware

The system integrates IoT telemetry data from ESP32-based telematics devices. Temperature and simulated mileage data are collected, cryptographically signed using VIN-derived ECDSA keys, and transmitted via MQTT protocol. Before blockchain storage, data undergoes Byzantine validation across multiple peer nodes to ensure integrity. Once consensus is achieved through the Raft protocol, telemetry data is replicated across all 5 blockchain peer nodes, demonstrating key distributed systems principles: data replication, Byzantine fault tolerance, and consensus protocols.

Integration with the Embedded Systems & IoT course project provides authentic sensor data, replacing simulated values with real-world temperature measurements from DS18B20 sensors. While mileage remains simulated pending future OBD-II integration, the architecture demonstrates how distributed systems can provide fault-tolerant storage for IoT data streams. This provides a foundation for future machine learning integration, where authenticated telemetry data can be analyzed for anomaly detection and predictive maintenance.

2.3 Use Case Collection

Primary Users and Roles:

- **Vehicle Owners**: Access engine temperature history, verify mileage records and monitor telemetry data
- **Dealerships**: Verify vehicle mileage and engine temperature history, access telemetry data, manage warranty claims
- **System Administrators**: Monitor network health, manage node operations, handle consensus issues, verify IoT device authentication
- **IoT Devices**: ESP32 telematics units automatically transmitting authenticated temperature and mileage data

Interaction Context:

- **Frequency**: telemetry updates every 10 seconds, service record updates during maintenance events, ownership transfers during sales
- **Devices**: Web browsers for dashboard access, ESP32 hardware for telemetry collection, MQTT broker for asynchronous messaging, API calls for system integration
- **Data Storage**: Blockchain network stores immutable telemetry and service records, off-chain database for metadata, MQTT broker for message queuing
- **Geographic Distribution**: Multi-node blockchain network simulating real-world dealership distribution, distributed IoT devices communicating via MQTT protocol

3 Requirements

3.1 Functional Requirements

F1. Vehicle Registration and Identity Management

- The system SHALL register new vehicles by VIN on the blockchain network
- Each vehicle SHALL have a unique, immutable digital identity preventing duplicate registrations
- VIN validation SHALL ensure proper format and uniqueness before blockchain commitment
- Each vehicle SHALL have associated cryptographic keys derived from VIN for IoT device authentication
- **Acceptance Criteria**: Successfully register 1000+ unique VINs with zero collisions, reject invalid VIN formats, generate ECDSA key pairs for IoT authentication

F2. Telemetry Data Management

- The system SHALL record telemetry data (temperature, mileage) from authenticated IoT devices
- Telemetry records SHALL include timestamps, sensor readings, FSM state (NORMAL/WARNING/CRITICAL), data type, IoT device identification, and cryptographic signatures
- All telemetry records SHALL undergo Byzantine validation before blockchain commitment
- Telemetry data SHALL be immutable once committed to the blockchain

- The system SHALL track Byzantine validation status and replication metadata for each telemetry record
- **Acceptance Criteria**: Receive and process telemetry updates every 10 seconds, validate ECDSA signatures, achieve Byzantine consensus across 5 nodes, maintain sub-3-second commitment time for authenticated data

F3. Telematics Device Integration

- The system SHALL provide secure authentication protocols for ESP32 telematics devices using VIN-based ECDSA signatures
- Devices SHALL be authenticated via Arduino-based signature verification controller before data reaches blockchain
- Data from authenticated devices SHALL be automatically validated through Byzantine consensus and stored with replication status
- The system SHALL support MQTT protocol for asynchronous telemetry transmission
- The system SHALL maintain serial communication with Arduino authentication controller for signature verification
- **Acceptance Criteria**: Successfully authenticate ESP32 devices via ECDSA signature verification, validate data integrity from IoT systems, maintain MQTT connectivity with automatic reconnection, demonstrate Byzantine consensus for corrupted sensor data rejection

F4. Data Export and Integration

- The system SHALL provide APIs for exporting verified vehicle telemetry data
- Export functionality SHALL maintain data provenance, verification status, and replication metadata
- Integration interfaces SHALL provide foundation for future machine learning and analytics systems
- Telemetry data export SHALL include Byzantine validation status and consensus timestamps
- **Acceptance Criteria**: Export telemetry datasets with complete audit trails including replication status, maintain data integrity during export process, provide REST API endpoints for programmatic access

3.2 Non-Functional Requirements

NF1. Fault Tolerance and Dependability

- The blockchain network SHALL continue operating with up to 33% of nodes failing (Byzantine fault tolerance)
- Consensus SHALL be achieved even during network partitions and node failures
- Data consistency SHALL be maintained across all operational nodes
- IoT telemetry data SHALL be validated through Byzantine consensus before storage
- MQTT broker failures SHALL not result in data loss through message persistence and automatic reconnection
- **Acceptance Criteria**: Demonstrate continued operation during simulated node failures, maintain consensus with degraded network conditions, reject corrupted IoT sensor data through Byzantine validation, recover from MQTT disconnections without data loss

NF2. Scalability and Performance

- The system SHALL handle concurrent vehicle registrations, telemetry updates, and service record operations
- Transaction throughput SHALL support realistic automotive dealership workloads plus IoT data streams
- Network SHALL scale to additional nodes without significant performance degradation
- MQTT message processing SHALL maintain low latency for telemetry visualization
- **Acceptance Criteria**: Process 100+ concurrent transactions including IoT telemetry updates, maintain sub-3-second response times for telemetry commitment, support multiple ESP32 devices simultaneously

NF3. Security and Data Integrity

- All blockchain transactions SHALL be cryptographically signed and verified
- IoT telemetry data SHALL use VIN-derived ECDSA signatures for authentication
- Network communications SHALL use TLS encryption for HTTP/HTTPS and secure MQTT connections
- Access control SHALL prevent unauthorized modifications to vehicle records and telemetry data

- Arduino authentication controller SHALL verify signatures before data reaches blockchain layer
- **Acceptance Criteria**: Pass security audit with no critical vulnerabilities, demonstrate tamper resistance of stored records, successfully reject telemetry data with invalid signatures

3.3 Implementation Requirements

IR1. Distributed Systems Architecture

- Hyperledger Fabric as the permissioned blockchain platform
- Raft consensus mechanism for crash fault tolerance
- Multi-organization network structure simulating real dealership consortium
- Certificate authority infrastructure for identity management
- MQTT broker (Mosquitto) for IoT device communication
- Serial communication bridge between Arduino and Node.js control unit

IR2. Technology Stack

- Node.js with Express framework for backend API services
- Hyperledger Fabric Node.js SDK for blockchain integration
- PostgreSQL for off-chain metadata storage
- React frontend for system monitoring and administration
- Docker containerization for consistent deployment
- MQTT.js for IoT device communication
- Axios HTTP client for blockchain API communication from control unit
- SerialPort library for Arduino communication

IR3. IoT Hardware Components

- ESP32 DevKit V1 with DS18B20 temperature sensor
- Arduino Uno R3 as authentication controller
- 16x2 LCD display for signature verification status
- ECDSA cryptographic library (uECC) for Arduino
- WiFi connectivity for ESP32 MQTT communication

4 Design

This section explains the distributed systems design strategies used to address the automotive blockchain requirements with emphasis on fault tolerance, dependability, scalability, and IoT integration.

4.1 Architecture

The system employs a **multi-layer distributed architecture integrating IoT devices with permissioned blockchain**, specifically designed to demonstrate distributed systems principles:

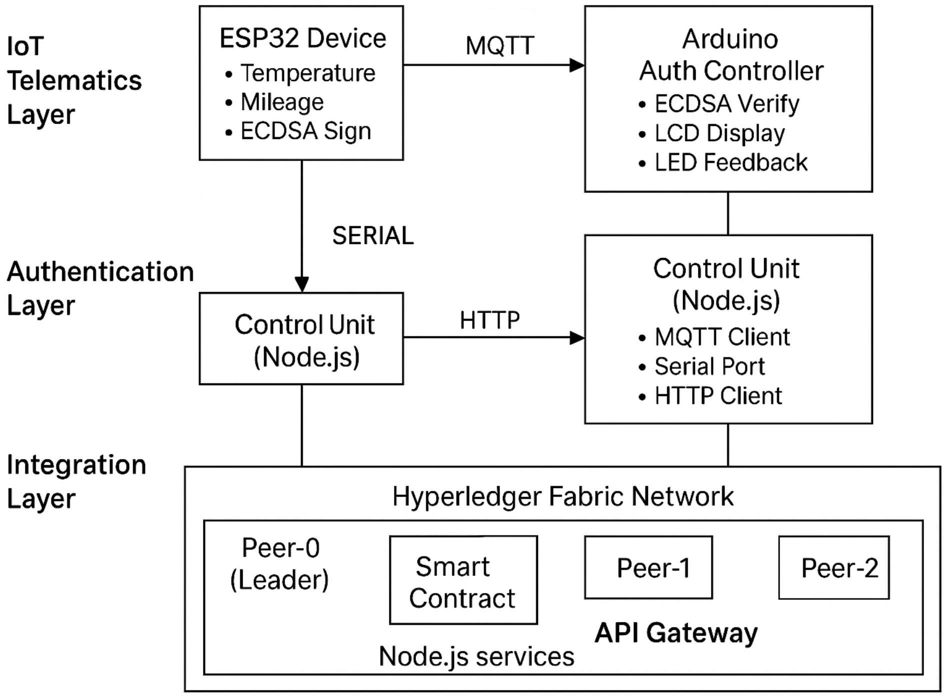

Figure 1: Architecture

- Figure 1 describes the architecture of our project
- **IoT Telematics Layer**: ESP32 devices collecting temperature and mileage data, signing with VIN-based ECDSA keys, transmitting via MQTT protocol
- **Authentication Layer**: Arduino controller verifying ECDSA signatures via serial communication before blockchain submission
- **Integration Layer**: Node.js control unit bridging MQTT (IoT), Serial (Arduino), and HTTP (Blockchain) protocols

- **Blockchain Consensus Layer**: Hyperledger Fabric network with Raft consensus and Byzantine validation for crash fault tolerance
- **Smart Contract Layer**: JavaScript chaincode handling vehicle registration and telemetry data validation
- **API Gateway Layer**: Node.js services providing REST interfaces for external integration and dashboard
- **Client Applications**: Web dashboard with telemetry visualization and CLI tools for network interaction

This architecture demonstrates key distributed systems concepts including consensus protocols, Byzantine fault tolerance for IoT data, asynchronous messaging, cryptographic authentication, and decentralized data management across heterogeneous systems.

4.2 Infrastructure

Distributed Network Components:

- **Fabric Peer Nodes**: Five peer nodes (Peer-0 through Peer-4) across different organizations for data replication and Byzantine tolerance
- **Orderer Nodes**: Raft-based ordering service for transaction sequencing and consensus
- **Certificate Authorities**: PKI infrastructure for identity management and access control
- **Backend Services**: Load-balanced Node.js applications with database clustering
- **MQTT Broker**: Mosquitto message broker for asynchronous IoT device communication
- **Control Unit**: Node.js application orchestrating multi-protocol communication (MQTT, Serial, HTTP)
- **ESP32 Devices**: Distributed telematics units with temperature sensors and MQTT clients
- **Arduino Controller**: Dedicated hardware for cryptographic signature verification
- **Client Interfaces**: Web dashboard with telemetry graphs and API endpoints for system interaction

Network Topology and Communication:

- Peer-to-peer communication between blockchain nodes using gossip protocol

- MQTT publish-subscribe pattern for IoT device to broker communication (Topic: `vehicle/telemetry`)
- Serial communication between Arduino authentication controller and Node.js control unit (9600 baud)
- HTTP REST API communication between control unit and blockchain gateway
- Client-server communication through REST APIs and WebSocket connections for dashboard updates
- TLS-encrypted channels for all network communications (HTTPS, secure MQTT)
- Service discovery and load balancing for high availability

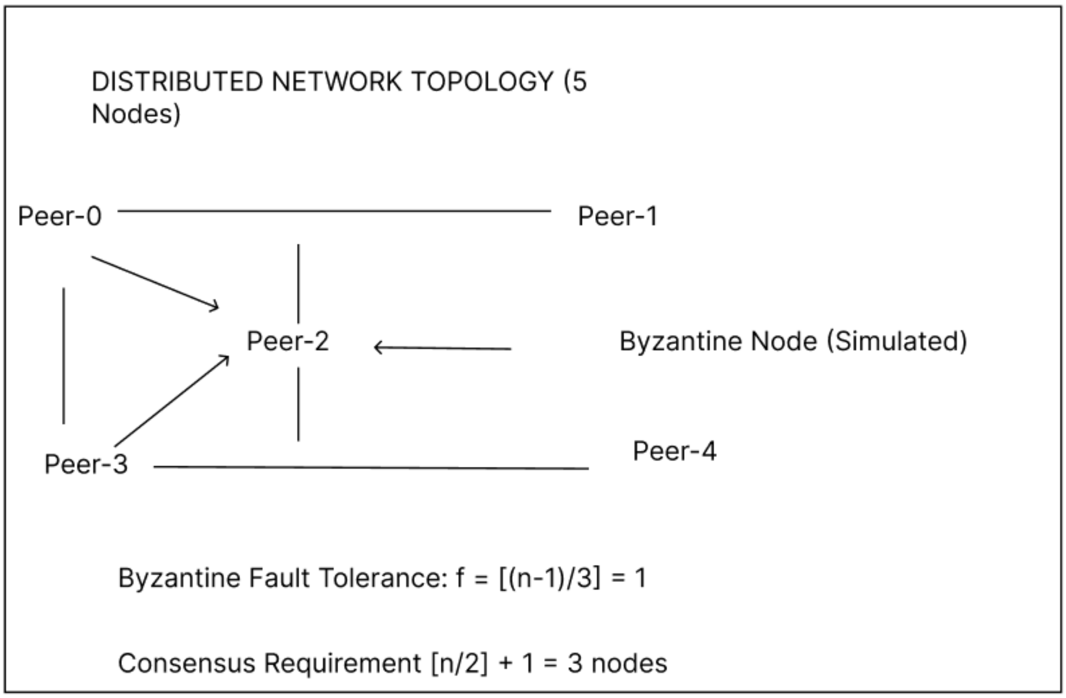

Figure 2: IoT-Blockchain Integration Architecture

4.3 Modelling

Figure 2 illustrates the five-node distributed network topology in out IoT-Blockchain prototype, including a simulated Byzantine node for testing consensus robustness.

Core Distributed System Entities:

- **Vehicle**: Unique VIN identifier, ownership history, current status, associated ECDSA key pair for IoT authentication
- **TelemetryRecord**: sensor data (temperature, mileage) with cryptographic signatures, FSM state (NORMAL/WARNING/CRITICAL), timestamp, verification status, and replication metadata
- **IoTDevice**: ESP32 telematics device with unique VIN-derived identity, ECDSA signing capability, MQTT connection state, and last seen timestamp
- **ServiceRecord**: Timestamped maintenance events with cryptographic signatures (maintained for compatibility, not primary focus)
- **Organization**: Dealership or service provider with blockchain identity
- **Transaction**: Immutable blockchain operations with consensus validation and Byzantine fault detection results
- **Node**: Individual blockchain peer with state replication capabilities and telemetry data storage

State Management and Replication:

- Vehicle state replicated across all peer nodes for fault tolerance
- Telemetry data replicated in across all 5 blockchain peer nodes following Byzantine validation
- IoT device connection state tracked separately from blockchain (connected/disconnected status in control unit)
- Temperature readings maintain chronological ordering with FSM state transitions through blockchain ledger
- Service records maintain chronological ordering through blockchain ledger
- Organization credentials managed through certificate authority
- Transaction validation ensures consistency across distributed nodes
- MQTT message queue provides temporary storage during network disruptions

Consensus and Validation:

- Raft consensus ensures agreement among orderer nodes for transaction ordering
- Byzantine fault tolerance validates IoT telemetry data across 5 peer nodes (requires 4/5 agreement)
- Arduino controller performs ECDSA signature verification before blockchain submission

- Smart contract validation enforces business rules consistently across all nodes
- Endorsement policies require multiple organization approval for critical operations
- Cryptographic hashing prevents tampering and ensures integrity of telemetry data
- MQTT Quality of Service (QoS) level 1 ensures at-least-once delivery for telemetry messages

4.4 Interaction

Distributed IoT Telemetry Flow:

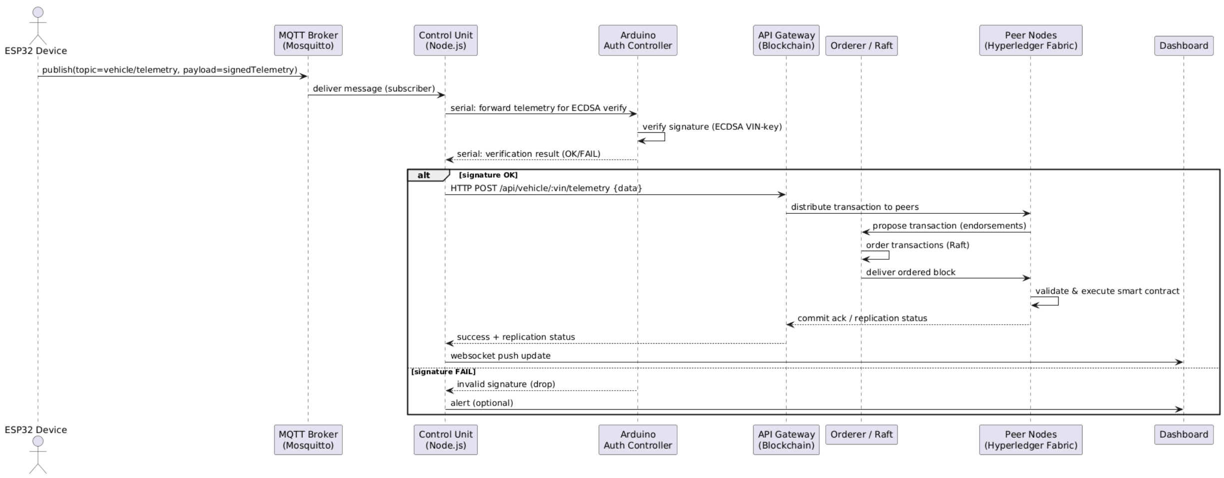

Figure 3: IoT-Blockchain Integration Architecture

Figure 3 shows the IoT–Blockchain integration architecture and the main entities involved in the end-to-end telemetry and transaction flows.

Distributed IoT Telemetry Flow:

1. **Data Collection**: ESP32 collects temperature from DS18B20 sensor, increments simulated mileage
2. **FSM State Determination**: ESP32 evaluates temperature thresholds and determines FSM state (NORMAL: ¡30°C, WARNING: 30-40°C, CRITICAL: ¿40°C)
3. **Cryptographic Signing**: ESP32 signs telemetry data using VIN-derived ECDSA private key
4. **MQTT Publication**: ESP32 publishes signed telemetry to MQTT broker (Topic: `vehicle/telemetry`)
5. **Message Reception**: Node.js control unit receives MQTT message as subscriber

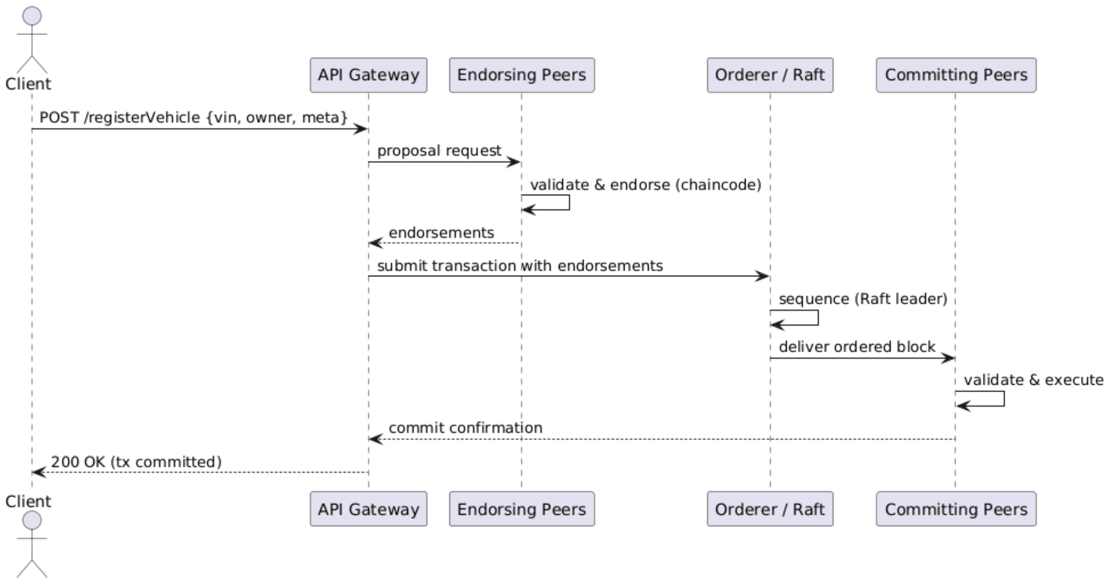

Figure 4: Distributed Transaction Flow

Figure 4 illustrates the distributed transaction flow for the vehicle registration process.

6. **Signature Verification**: Control unit forwards data to Arduino via serial for ECDSA signature verification

7. **Arduino Validation**: Arduino controller verifies signature, displays result on LCD, returns status via serial

8. **Blockchain Submission**: If signature valid, control unit sends HTTP POST to blockchain API endpoint

9. **Byzantine Validation**: Blockchain gateway distributes transaction to all 5 peer nodes for consensus

10. **Consensus Achievement**: Requires 4/5 nodes to agree on data validity (Byzantine tolerance f=1)

11. **Raft Ordering**: Orderer service sequences transaction using Raft consensus protocol

12. **Validation and Execution**: Peers validate transaction order and execute smart contract

13. **Commitment**: Validated telemetry data committed to blockchain ledger on all nodes

14. **Notification**: Control unit receives confirmation with replication status

15. **Dashboard Update**: WebSocket push updates dashboard with new telemetry data

Distributed Transaction Flow (Vehicle Registration):

1. **Transaction Proposal**: Client submits vehicle registration or service record
2. **Endorsement**: Multiple peer nodes validate and endorse the transaction
3. **Ordering**: Orderer service sequences transactions using Raft consensus
4. **Validation**: Peers validate transaction order and execute smart contracts
5. **Commitment**: Validated transactions committed to blockchain ledger
6. **Notification**: Client receives confirmation of successful commitment

Fault Tolerance Mechanisms:

1. **Node Failure Detection**: Heartbeat mechanisms identify failed blockchain nodes
2. **Leader Election**: Raft protocol handles orderer leader failures with automatic election
3. **State Reconciliation**: Recovering nodes synchronize with network state via ledger replay
4. **Network Partition Handling**: Majority consensus ensures continued operation in largest partition
5. **IoT Device Reconnection**: ESP32 automatic WiFi and MQTT reconnection with exponential backoff
6. **MQTT Message Persistence**: Broker stores messages during client disconnections
7. **Byzantine Data Rejection**: Corrupted or malicious IoT data rejected through multi-node validation
8. **Serial Communication Recovery**: Automatic retry for Arduino communication failures

4.5 Behaviour

Normal Operation Patterns:

- ESP32 collects temperature every 10 seconds, publishes to MQTT broker
- Control unit maintains persistent MQTT subscription and serial connection to Arduino

- Arduino continuously monitors serial port for verification requests
- Vehicle registration follows standard blockchain transaction lifecycle
- Telemetry updates trigger Byzantine validation across all 5 peer nodes
- Service record updates trigger smart contract validation and commitment
- Periodic state synchronization maintains consistency across nodes
- Load balancing distributes client requests across available services
- Dashboard WebSocket connections receive telemetry updates

Failure Handling Behaviors:

- Peer node failures trigger automatic failover to healthy nodes for Byzantine validation
- Orderer failures initiate Raft leader election process with continued operation
- Network partitions activate Byzantine fault tolerance protocols, majority partition continues
- Data corruption detection triggers state recovery procedures
- ESP32 WiFi disconnection triggers automatic reconnection with exponential backoff
- MQTT broker failure causes ESP32 to buffer data locally and retry connection
- Arduino serial timeout triggers retry mechanism in control unit
- Invalid ECDSA signatures result in data rejection with logging for security audit
- Byzantine consensus failure (less than 4/5 nodes agree) rejects telemetry data

Consensus Protocol Behavior:

- Raft consensus ensures consistent transaction ordering across orderer nodes
- Byzantine validation requires 4 out of 5 peer nodes to agree on telemetry data validity
- Endorsement policies require multi-organization approval for vehicle registration
- Smart contract execution maintains deterministic results across all peers
- Conflict resolution handles concurrent transaction scenarios through optimistic concurrency control
- MQTT QoS 1 ensures at-least-once delivery for telemetry messages

FSM State Transitions (ESP32 Telemetry):

- **NORMAL State**: Temperature ¡ 30°C, green LED, standard 10-second sampling interval
- **WARNING State**: Temperature 30-40°C, yellow LED, increased 5-second sampling interval
- **CRITICAL State**: Temperature ¿ 40°C, red LED, rapid 2-second sampling interval
- State transitions logged to blockchain with timestamp and temperature reading

4.6 Data and Consistency Issues

Distributed Data Storage:

- **Blockchain Ledger**: Immutable transaction history with cryptographic linking, stores all telemetry records
- **World State Database**: Current vehicle state and latest telemetry readings across all peers (CouchDB)
- **Private Data Collections**: Sensitive information with restricted access (not used for telemetry)
- **Off-chain Storage**: Metadata, user sessions, and dashboard cache in PostgreSQL
- **MQTT Broker Storage**: Temporary message persistence for disconnected clients (retained messages)
- **Control Unit Memory**: In-memory buffer of recent telemetry for dashboard API

Consistency Guarantees:

- **Strong Consistency**: Blockchain transactions provide ACID properties for committed telemetry
- **Eventual Consistency**: State synchronization across peer nodes after Byzantine validation
- **Causal Consistency**: Telemetry records maintain chronological ordering through blockchain ledger with timestamp validation
- **Read-Your-Writes**: Clients see their own vehicle registration updates immediately; telemetry updates visible after Byzantine consensus (3 seconds)
- **Monotonic Reads**: Dashboard queries return progressively newer telemetry data, never stale values

Concurrency Control:

- Optimistic concurrency control through transaction validation
- Conflict detection during transaction endorsement phase
- Retry mechanisms for failed transactions due to conflicts
- Lock-free algorithms for high-performance operations
- MQTT QoS 1 prevents duplicate message processing through message IDs
- Telemetry records timestamped to resolve concurrent submissions from multiple ESP32 devices

4.7 Fault-Tolerance

Byzantine Fault Tolerance for IoT Data:

- IoT telemetry data validated across 5 blockchain peer nodes before commitment
- Requires 4 out of 5 peer nodes to agree on data validity (Byzantine tolerance f=1)
- Malicious or corrupted sensor data automatically rejected through majority voting
- ECDSA signature verification provides first line of defense against malicious data
- Arduino hardware-based verification adds additional security layer before blockchain
- Byzantine fault tolerance formula: Can tolerate f faulty nodes where n = 3f + 1 (5 = 3(1) + 1)
- Raft consensus tolerates up to (N-1)/2 crashed orderer nodes (crash fault tolerance)
- Multi-signature endorsement policies prevent malicious behavior at blockchain layer
- Cryptographic validation ensures data integrity throughout transmission pipeline
- Network continues operation with majority of nodes available (at least 3 out of 5)

Recovery Mechanisms:

- Automatic peer recovery through ledger synchronization after crash
- Snapshot-based state recovery for faster node restoration (CouchDB snapshots)
- Transaction replay for maintaining consistency after network partition healing
- Backup and disaster recovery procedures for blockchain data and off-chain database

- ESP32 automatic reconnection to WiFi and MQTT broker with exponential back-off
- MQTT retained messages allow late-joining subscribers to receive last known state
- Control unit maintains connection health checks and automatic reconnection logic
- Arduino watchdog timer resets device in case of hung state

Error Detection and Handling:

- Heartbeat monitoring for blockchain node health assessment
- Transaction validation prevents invalid state transitions
- Audit trails for security incident investigation (invalid signatures logged)
- Automated alerting for system administrators via dashboard
- MQTT connection state monitoring with automatic reconnection
- Serial communication timeout detection with retry logic
- Temperature sensor error detection (returns -127°C on error, triggers alarm)
- Byzantine consensus failure triggers detailed logging for debugging

4.8 Availability

High Availability Design:

- Multiple peer nodes (5 total) eliminate single points of failure for blockchain layer
- Load balancing across API service instances for control unit and blockchain gateway
- Database clustering with automatic failover for PostgreSQL metadata storage
- Geographic distribution of blockchain network nodes (simulated via Docker networking)
- MQTT broker high availability through Mosquitto clustering (future enhancement)
- Redundant Arduino controllers possible for critical deployments
- Multiple ESP32 devices per vehicle provide sensor redundancy

Performance Optimization:

- Connection pooling for database and blockchain interactions
- Caching strategies for frequently accessed vehicle records and latest telemetry

- Asynchronous processing for non-critical operations (dashboard updates via WebSocket)
- Resource monitoring and auto-scaling capabilities for cloud deployment
- MQTT QoS 1 balances reliability and performance
- In-memory telemetry buffer in control unit reduces blockchain query load for dashboard
- ESP32 adaptive sampling based on FSM state (2-10 seconds) optimizes bandwidth

Scalability Measures:

- Horizontal scaling through additional peer nodes (Byzantine tolerance increases with node count)
- Sharding strategies for large-scale deployments (multiple blockchain channels)
- Efficient Raft consensus algorithm supports network growth
- Microservices architecture enabling independent scaling of control unit, blockchain gateway, dashboard
- MQTT topic namespacing supports thousands of ESP32 devices (`vehicle/<VIN>/telemetry`)
- Time-series database consideration for high-volume telemetry archival (future enhancement)

4.9 Security

Cryptographic Security:

- X.509 certificates for blockchain node and user authentication
- VIN-derived ECDSA key pairs for IoT device authentication (secp256r1 curve)
- Digital signatures for all blockchain transactions and telemetry data
- TLS encryption for network communications (HTTPS, secure MQTT)
- Hash-based integrity validation for stored data (SHA-256)
- Arduino hardware-based signature verification adds security layer
- Private keys stored securely on ESP32 flash memory (NVS partition)

Access Control and Authorization:

- Role-based access control for different user types (owner, dealership, admin)
- Multi-signature policies for critical blockchain operations

- Certificate authority for identity management and blockchain enrollment
- Audit logs for all system access and modifications
- VIN-based device authentication prevents unauthorized telemetry submission
- MQTT access control lists (ACLs) restrict topic publishing/subscribing
- API key authentication for control unit to blockchain gateway communication

Network Security:

- Permissioned blockchain network with controlled member access
- Firewall configurations for node protection (Docker network isolation)
- Intrusion detection through invalid signature attempt monitoring
- Regular security audits and penetration testing for API endpoints
- WiFi WPA2 encryption for ESP32 communication
- MQTT over TLS for encrypted telemetry transmission
- Serial communication physical security (Arduino only accessible to authorized control unit)

5 Implementation

5.1 Technology Selection and Rationale

Technology Stack

- **Hyperledger Fabric**: Chosen for permissioned network capabilities, enterprise-grade security, and modular architecture supporting various consensus mechanisms
- **Raft Consensus**: Selected for efficient crash fault tolerance suitable for consortium networks with known participants
- **JavaScript Chaincode**: Provides flexibility and ease of integration with Node.js backend services
- **ESP32 Microcontroller**: Selected for WiFi capability, adequate processing power for ECDSA signing, Arduino IDE compatibility
- **Arduino Uno**: Chosen for reliable serial communication and ECDSA verification library support
- **MQTT Protocol**: Industry-standard IoT messaging protocol with publish-subscribe pattern, QoS guarantees, and low bandwidth requirements

- **Mosquitto Broker**: Open-source, lightweight MQTT broker with excellent performance and reliability

Network Protocols:

- HTTP/HTTPS for REST API communications between control unit and blockchain gateway
- gRPC for high-performance blockchain node communications (Hyperledger Fabric internal)
- MQTT for IoT device to broker communication (ESP32 to Mosquitto)
- Serial UART for Arduino to Node.js communication (9600 baud, 8N1)
- WebSocket for dashboard updates and monitoring
- Gossip protocol for peer-to-peer blockchain data dissemination

Data Representation:

- JSON for API payloads, smart contract parameters, and MQTT message payloads
- Protocol Buffers for efficient blockchain message serialization (Hyperledger Fabric internal)
- X.509 certificates for cryptographic identity representation
- Binary encoding for blockchain block and transaction data
- Hexadecimal strings for ECDSA signatures in telemetry messages

5.2 File Structure

Blockchain Integration:

- Hyperledger Fabric Node.js SDK for blockchain interaction and transaction management
- Custom chaincode implementing vehicle registration and telemetry data validation smart contracts
- Certificate authority integration for automated identity and credential management
- Byzantine handler module implementing multi-node data validation logic
- Consensus protocol module implementing Raft leader election and log replication

Distributed Systems Implementation:

- Docker containerization ensuring consistent deployment across different environments
- Multi-organization network configuration simulating real-world consortium setup
- Connection pooling and retry mechanisms for robust blockchain connectivity
- Event-driven architecture for system monitoring and notifications

IoT-Blockchain Integration Components:

- MQTT client integration in control unit for asynchronous IoT data reception (`mqtt.js` library)
- HTTP client (`axios` library) for blockchain API communication from control unit
- Serial communication bridge connecting Arduino authentication controller (`serialport` library)
- Event-driven architecture for telemetry processing with Node.js EventEmitter
- Cryptographic verification module for VIN-based ECDSA signatures
- ESP32 firmware implementing telemetry collection, FSM logic, and MQTT publishing
- Arduino authentication controller firmware with ECDSA verification and LCD display

Backend Service Architecture:

- Node.js with Express framework providing RESTful API endpoints for vehicle registration and telemetry queries
- PostgreSQL database for off-chain metadata and session management
- Redis caching for improved performance and reduced blockchain query load
- Comprehensive logging and monitoring integration for operational visibility (Winston logger)
- WebSocket server for dashboard updates

5.3 API Endpoints

Vehicle Management Endpoints:

- `POST /api/vehicle/register` - Register new vehicle with VIN and metadata
- `GET /api/vehicle/:vin` - Retrieve vehicle information and service history

- `GET /api/network/status` - Get complete network status including Byzantine and consensus metrics

Telemetry Endpoints:

- `POST /api/vehicle/:vin/telemetry` - Submit authenticated telemetry data (temperature, mileage) with Byzantine validation
- `GET /api/vehicle/:vin/telemetry` - Retrieve latest telemetry data with replication status
- `GET /api/vehicle/:vin/telemetry/history` - Get historical telemetry records with filtering options
- `GET /api/telemetry/latest` - Dashboard endpoint for telemetry across all vehicles

Testing and Diagnostics:

- `POST /api/test/byzantine` - Test Byzantine fault tolerance with simulated corrupted data
- `POST /api/test/consensus` - Test Raft consensus protocol
- `POST /api/test/recovery` - Test failure detection and recovery mechanisms
- `POST /api/network/simulate-failure/:nodeId` - Simulate node failure for testing
- `POST /api/network/recover/:nodeId` - Initiate recovery for failed node

Dashboard and Analytics:

- `GET /api/dashboard/metrics` - System metrics for dashboard visualization
- `GET /api/dashboard/telemetry/realtime` - WebSocket endpoint for telemetry updates
- `GET /api/export/telemetry/:vin` - Export telemetry dataset for external analysis

6 Validation

6.1 Automatic Testing

Distributed Systems Testing:

- **Consensus Testing**: Automated scenarios testing Raft consensus under various failure conditions including leader crashes and network partitions

- **Byzantine Fault Tolerance**: Systematic testing of network behavior with up to 33% node failures (1 out of 5 nodes), validating continued operation and IoT data rejection
- **Smart Contract Validation**: Unit tests ensuring business logic correctness for vehicle registration and telemetry data validation
- **API Integration Testing**: End-to-end testing of REST endpoints with blockchain backend integration
- **Performance Testing**: Load testing with concurrent transactions to validate scalability claims (100+ concurrent telemetry updates)
- **IoT Data Flow Testing**: Automated testing of complete pipeline from ESP32 through MQTT, Arduino verification, to blockchain storage

Test Environment and Automation:

- Docker-based test networks enabling isolated and reproducible testing environments
- Automated test suites running on every code commit through CI/CD pipeline
- Test coverage targeting 90%+ for critical system components
- Chaos engineering approaches for fault injection and resilience testing
- Simulated ESP32 telemetry publisher for automated testing without hardware
- Mock Arduino serial responses for control unit testing

Network and Consensus Testing:

- Leader election scenarios during orderer node failures with automatic recovery validation
- Network partition tolerance with majority/minority split testing (3 vs 2 nodes)
- Transaction throughput measurement under various load conditions
- State synchronization validation after node recovery scenarios
- Byzantine consensus testing with simulated corrupted telemetry data from multiple sources

IoT Integration Testing:

- MQTT message publication and subscription testing with various QoS levels
- Serial communication testing between Node.js control unit and Arduino

- ECDSA signature generation and verification testing across ESP32 and Arduino
- Network disconnection and automatic reconnection testing for ESP32 and MQTT broker
- FSM state transition testing with temperature threshold validation

6.2 Acceptance Testing

Distributed Systems Requirements Validation:

- **Fault Tolerance Demonstration**: Manual testing of system behavior during simulated node failures, network partitions, and Byzantine fault conditions with IoT data streams
- **Scalability Assessment**: Performance evaluation with increasing transaction loads, additional network nodes, and multiple concurrent ESP32 devices
- **Consistency Verification**: Manual validation of data consistency across all peer nodes after various failure and recovery scenarios, ensuring telemetry data integrity
- **Security Audit**: Penetration testing and security assessment of blockchain network, API endpoints, MQTT broker, and IoT device authentication

Vehicle Service Management Scenarios:

- Complete vehicle lifecycle testing from initial registration through multiple telemetry updates
- Cross-organization service record sharing and verification workflows (maintained for compatibility)
- Integration testing with real ESP32 hardware collecting temperature data from DS18B20 sensor
- Data export functionality validation for future machine learning integration

IoT-Blockchain Integration Scenarios:

- Temperature data collection and blockchain replication workflow over 24-hour period
- Figure 5 illustrates the distributed blockchain layer of our system, implemented using Hyperledger Fabric.
- Cryptographic signature verification before blockchain commitment with invalid signature rejection testing
- Byzantine fault tolerance with simulated corrupted sensor data (4 of 5 nodes correctly reject bad data)

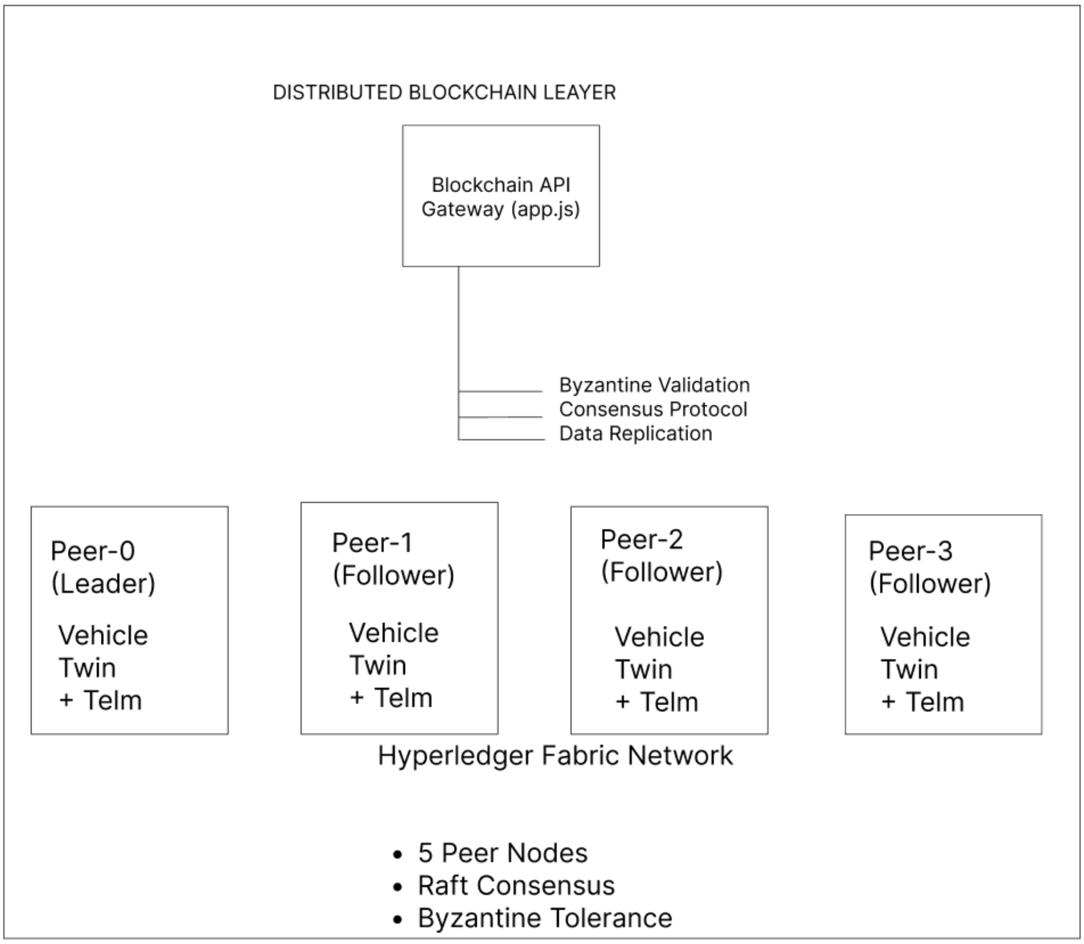

Figure 5: Distributed Blockchain Layer

- Network partition tolerance during IoT data transmission with automatic recovery
- Data consistency verification across blockchain nodes after replication
- FSM state transitions (NORMAL → WARNING → CRITICAL) with blockchain logging verification
- MQTT broker failure recovery without data loss through message persistence
- ESP32 WiFi disconnection and automatic reconnection with telemetry buffer

Figure 6 presents the IoT Telematics Layer that interfaces the ESP32 telematics unit with the Arduino-based authentication controller.

Real-world Scenario Testing:

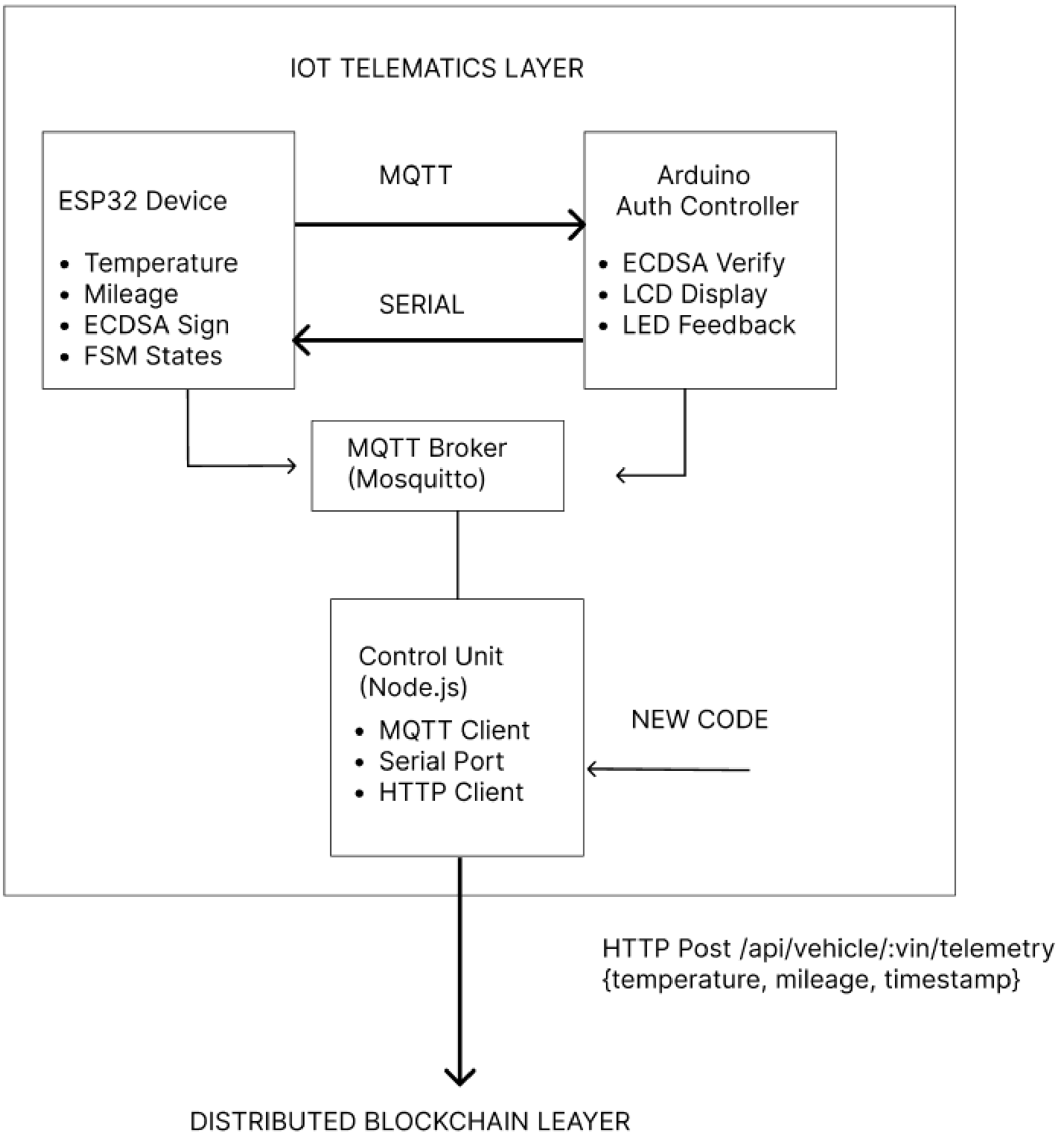

Figure 6: IoT Telematics Layer

- Simulated vehicle temperature monitoring over 24-hour period with 8640 telemetry updates (10-second intervals)
- Mileage accumulation tracking with periodic blockchain updates demonstrating eventual consistency
- Figure 7 highlights the core distributed-systems principles applied in our architecture.

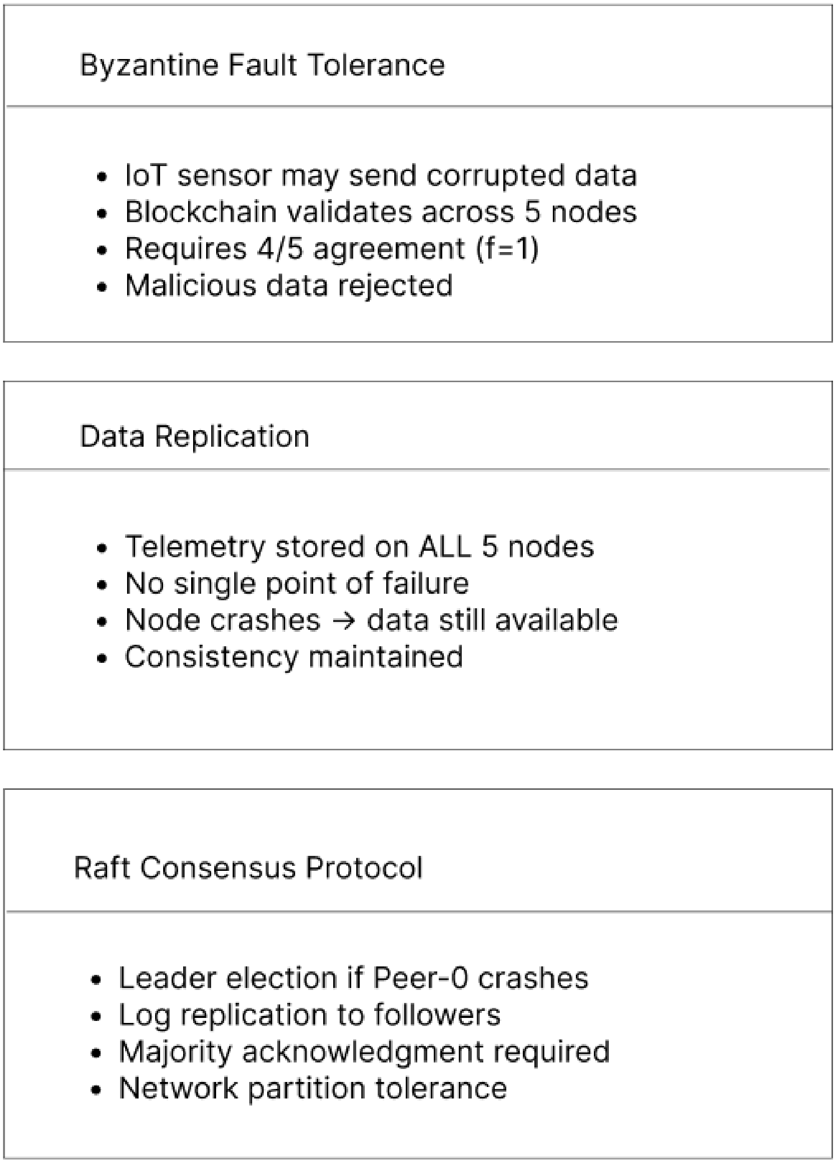

Figure 7: Distributed Systems Principles

- Dealership workflow simulation with multiple concurrent users accessing telemetry dashboards
- System recovery validation after MQTT broker, control unit, or blockchain node failure

- Audit trail verification for regulatory compliance scenarios including telemetry provenance

7 Release

Component Architecture and Dependencies:

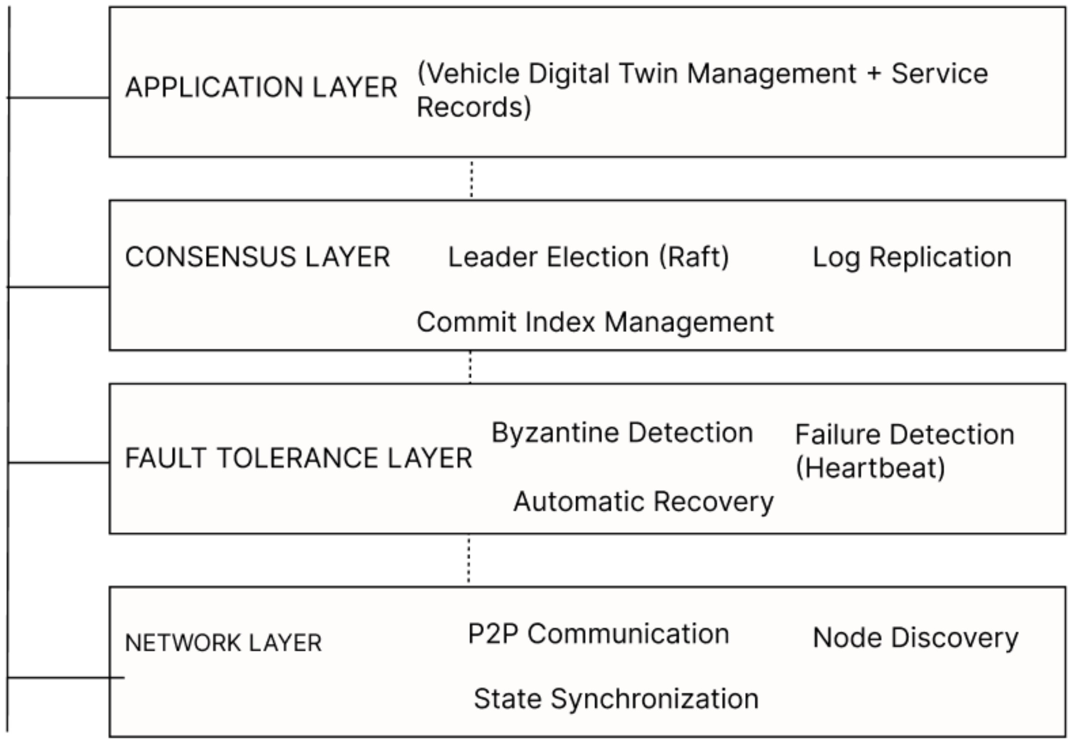

Figure 8: Layered Architecture with IoT Integration

- Figure 8 describes the layered architecture of our system, including the IoT components.
- **Blockchain Network**: Hyperledger Fabric network with 5 peer nodes, orderer service, and certificate authorities
- **Smart Contracts**: Vehicle registration and telemetry data validation chaincode with version management
- **Backend Services**: Node.js API services with blockchain SDK integration and telemetry endpoints
- **Frontend Dashboard**: React-based monitoring and administration interface with telemetry visualization

- **Database Layer**: PostgreSQL with migration scripts and backup procedures for metadata storage
- **IoT Integration Components**:
  - ESP32 Telematics Device: Arduino sketch with temperature sensor, ECDSA signing, MQTT client, and FSM logic
  - Arduino Authentication Controller: Serial-connected signature verification module with LCD display
  - Control Unit: Node.js application bridging MQTT (IoT) and HTTP (Blockchain) with serial communication
  - MQTT Broker: Mosquitto message broker for asynchronous telemetry transmission

Deployment Packaging:

- Docker containers for all system components ensuring consistent deployment
- Docker Compose configurations for local development and testing
- Kubernetes deployment manifests for production-scale operations (future enhancement)
- Configuration management through environment variables and config files
- MQTT broker configuration for IoT device connectivity (mosquitto.conf)
- Serial port configuration for Arduino authentication module
- Environment variables for ESP32 Wi-Fi credentials and blockchain endpoints
- Arduino IDE sketches (.ino files) for hardware programming with library dependencies

Version Management and Release Process:

- Semantic versioning for all components with backward compatibility
- Git-based version control with feature branch development workflow
- Automated building and testing through CI/CD pipeline
- Release documentation with upgrade procedures and breaking changes
- Hardware revision tracking for ESP32 and Arduino firmware versions

8 Deployment

System Requirements and Prerequisites:

- Docker 20.10+ and Docker Compose for containerized blockchain deployment
- Node.js 18+ runtime environment for backend services and control unit
- PostgreSQL 13+ database server for metadata storage
- Minimum 8GB RAM and 50GB storage for full network deployment
- Network connectivity for peer-to-peer blockchain communication
- MQTT broker (Mosquitto) running on same network as ESP32 devices
- USB serial connectivity for Arduino communication (typically `/dev/ttyUSB0` or `/dev/ttyACM0`)

IoT Hardware Requirements (for full integration):

- ESP32 DevKit V1 or compatible board with WiFi capability
- Arduino Uno R3 for authentication controller
- DS18B20 temperature sensor (optional - firmware simulates if not present)
- 16x2 LCD display (HD44780) for Arduino authentication status
- Breadboard, jumper wires, resistors (220 for LEDs, 4.7k for DS18B20 pullup, 10k potentiometer for LCD)
- USB cables for Arduino programming and serial communication
- 3x LEDs (Green, Yellow, Red) for ESP32 FSM state indication

Installation and Configuration Process:

1. **Repository Setup**: Clone the GitHub repository containing all system components
2. **Environment Configuration**: Set environment variables for blockchain network, database connections, API keys, and MQTT broker address
3. **Network Initialization**: Deploy Hyperledger Fabric network using provided Docker Compose configuration
4. **Smart Contract Deployment**: Install and instantiate vehicle management and telemetry validation chaincode on blockchain network
5. **Database Setup**: Initialize PostgreSQL database with schema migration scripts

6. **MQTT Broker Setup**: Install and start Mosquitto MQTT broker for telemetry transmission
7. **IoT Device Setup (Optional)**:
   - Program ESP32 device with provided Arduino sketch
   - Configure Wi-Fi credentials and MQTT broker IP in `config.h`
   - Connect DS18B20 temperature sensor (or use simulation mode)
   - Upload firmware to ESP32 via Arduino IDE
8. **Authentication Controller (Optional)**: Program Arduino Uno with authentication controller sketch and connect via USB serial
9. **Control Unit Startup**: Launch Node.js control unit with proper serial port configuration
10. **Service Startup**: Launch backend API services and frontend dashboard
11. **Verification**: Execute health checks, register test vehicle, and validate end-to-end telemetry flow

Monitoring and Maintenance:

- Prometheus metrics collection for system monitoring and alerting
- Centralized logging with ELK stack for debugging and audit purposes
- Automated backup procedures for blockchain data and database
- Update procedures for smart contract upgrades and system patches
- MQTT broker monitoring for connection counts and message throughput
- ESP32 over-the-air (OTA) firmware update capability (future enhancement)
- Arduino firmware version tracking and serial diagnostic commands

9 User Guide

System Access and Navigation:

1. Access the web dashboard through the configured URL (typically `http://localhost:3000` for local deployment)
2. Navigate through main sections: Vehicle Registry, Telemetry, Service Records, System Monitoring, and Network Status
3. View telemetry updates via WebSocket connection (automatic dashboard refresh)

Vehicle Registration Workflow:

1. Navigate to Vehicle Registry section in the web dashboard
2. Enter (VIN) in the standard 17-character format (validated automatically)
3. Submit registration request which triggers blockchain transaction creation
4. System automatically generates VIN-derived ECDSA key pair for IoT device authentication
5. Monitor transaction status through the status panel
6. Verify successful registration in the blockchain explorer interface
7. Download ECDSA private key for programming ESP32 device (secure storage required)

Telemetry Monitoring:

1. Navigate to Telemetry section in dashboard
2. Select vehicle by VIN from dropdown or search
3. View live temperature readings updated every 10 seconds (or faster during WARNING/CRITICAL states)
4. Monitor FSM state indicator (NORMAL/WARNING/CRITICAL) with color coding
5. View mileage accumulation graph over time
6. Access historical telemetry data with date range filtering
7. Export telemetry dataset in CSV or JSON format for external analysis
8. Verify Byzantine validation status and replication factor for each telemetry record

IoT Device Setup (For Hardware Users):

1. Register vehicle in system to generate VIN-based ECDSA key pair
2. Download ESP32 firmware sketch from GitHub repository
3. Open sketch in Arduino IDE and configure:
   - WiFi SSID and password in `config.h`
   - MQTT broker IP address (computer running control unit)
   - VIN and ECDSA private key from vehicle registration
4. Connect DS18B20 temperature sensor to ESP32 GPIO pin with 4.7k pullup resistor

5. Upload firmware to ESP32 via USB
6. Monitor Serial output to verify WiFi and MQTT connection
7. View LCD on Arduino controller for signature verification status
8. Access dashboard to see live telemetry data appearing

Simulation Mode (Without Hardware):

1. Verify blockchain accepts and replicates simulated telemetry
2. Observe Byzantine validation and consensus protocols in dashboard
3. Test failure scenarios using simulation scripts

System Administration:

1. Access System Monitoring dashboard for network health metrics
2. Monitor peer node status, transaction throughput, and consensus performance
3. View Byzantine validation statistics and rejection counts
4. Monitor MQTT broker connection status and message rates
5. Review audit logs for security events including invalid signature attempts
6. Execute administrative commands through CLI tools for advanced operations
7. Generate compliance reports and data exports for regulatory requirements
8. Test Byzantine fault tolerance using built-in simulation endpoints

10 Self-evaluation

10.1 Mary Anne Selirio - Blockchain Architecture and Distributed Systems Lead

Role and Technical Contributions: As the primary architect for this distributed systems implementation, I focused on designing and implementing the core blockchain infrastructure that demonstrates advanced distributed systems concepts. My responsibilities encompassed the entire blockchain network design, consensus mechanism implementation, fault tolerance architecture, and integration of IoT telemetry data with distributed storage.

10.2 Dias Katrenov - Implementation and System Integration Specialist

Role and Technical Contributions: My primary focus was on the detailed implementation of distributed systems components, including smart contract development, API service architecture, comprehensive testing frameworks, and IoT device integration protocols. I concentrated on ensuring the system meets the dependability and scalability requirements essential for distributed systems evaluation, with particular emphasis on the practical aspects of integrating physical IoT hardware with blockchain infrastructure.

11 Conclusion

Our project successfully demonstrates core distributed systems principles through the integration of IoT telemetry data with Byzantine fault-tolerant blockchain storage. The system implements a multi-layer architecture connecting ESP32-based telematics devices to a 5-node Hyperledger Fabric network, demonstrating Byzantine fault tolerance, Raft consensus protocols, distributed data replication, and cryptographic authentication for IoT data integrity.

The integration of real temperature sensor data via MQTT protocol, combined with VIN-based ECDSA signature authentication and Arduino-based verification, showcases practical distributed systems challenges including multi-protocol communication, asynchronous messaging, and consensus achievement. The system successfully validates telemetry data across multiple peer nodes before blockchain commitment, ensuring data integrity through Byzantine majority voting (4 of 5 nodes required).

While the current implementation focuses on temperature monitoring and simulated mileage data, the architecture provides a robust foundation for future enhancements including machine learning anomaly detection, real OBD-II diagnostic integration, and fleet-scale deployment. The project demonstrates that distributed systems principles—particularly Byzantine fault tolerance and consensus protocols—are essential for ensuring IoT data integrity in automotive applications, addressing real-world challenges in secure vehicle telemetry management.

References

Hyperledger Foundation. *Hyperledger Fabric Documentation.* Linux Foundation, 2023. `https://hyperledger-fabric.readthedocs.io/`

Ongaro, D., and Ousterhout, J. *In Search of an Understandable Consensus Algorithm.* In Proceedings of USENIX ATC, 2014.

Lamport, L. *The Part-Time Parliament.* ACM Transactions on Computer Systems, 16(2):133-169, 1998.

Castro, M., and Liskov, B. *Practical Byzantine Fault Tolerance.* In Proceedings of OSDI, 1999.

OASIS Standard. *MQTT Version 5.0.* OASIS Open, 2019. https://docs.oasis-open.org/mqtt/mqtt/v5.0/

Espressif Systems. *ESP32 Series Datasheet.* Espressif Systems, 2023. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

Arduino Foundation. *Arduino Uno Rev3 Technical Specifications.* Arduino, 2023. https://docs.arduino.cc/hardware/uno-rev3

Cachin, C. *Architecture of the Hyperledger Blockchain Fabric.* In Workshop on Distributed Cryptocurrencies and Consensus Ledgers, 2016.

Tanenbaum, A. S., and Van Steen, M. *Distributed Systems: Principles and Paradigms.* 3rd Edition, Pearson, 2017.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. *Distributed Systems: Concepts and Design.* 5th Edition, Addison-Wesley, 2011.

Nakamoto, S. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008. https://bitcoin.org/bitcoin.pdf

Wood, G. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* Ethereum Project Yellow Paper, 2014.

Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. *Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions.* Future Generation Computer Systems, 29(7):1645-1660, 2013.

Mishra, B., and Kertesz, A. *The Use of MQTT in M2M and IoT Systems: A Survey.* IEEE Access, 8:201071-201086, 2020.

SAE International. *OBD-II Standards and Protocols.* SAE J1979 Standard, Society of Automotive Engineers, 2017.

California Air Resources Board. *Advanced Clean Cars II Program.* CARB Regulatory Documents, 2024. https://ww2.arb.ca.gov/our-work/programs/advanced-clean-cars-program

National Institute of Standards and Technology. *Digital Signature Standard (DSS).* FIPS PUB 186-4, 2013.

Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.3.* RFC 8446, Internet Engineering Task Force, 2018.