```
import java.io.*;
import java.net.*;
import java.security.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.zip.*;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

// Custom exceptions for distributed system
class DistributedSystemException extends Exception {
    private final ErrorType errorType;

    public DistributedSystemException(String message, ErrorType errorType) {
        super(message);
        this.errorType = errorType;
    }

    public DistributedSystemException(String message, ErrorType errorType, Throwable cause) {
        super(message, cause);
        this.errorType = errorType;
    }

    public ErrorType getErrorType() { return errorType; }
}

enum ErrorType {
    NETWORK_ERROR, SERIALIZATION_ERROR, DESERIALIZATION_ERROR,
    CHECKSUM_MISMATCH, TIMEOUT, SECURITY_ERROR, CONNECTION_LOST
}

// Data packet for transmission
class DataPacket implements Serializable {
    private static final long serialVersionUID = 1L;

    private final UUID packetId;
    private final String sourceId;
    private final String destinationId;
    private final LocalDateTime timestamp;
    private final int sequenceNumber;
    private final int totalPackets;
    private final byte[] payload;
    private final String checksum;
    private final String encryptionHash;
    private transient boolean isAcknowledged;

    public DataPacket(String sourceId, String destinationId, int sequenceNumber,
                      int totalPackets, byte[] payload, String checksum, String encryptionHash) {
        this.packetId = UUID.randomUUID();
        this.sourceId = sourceId;
        this.destinationId = destinationId;
        this.timestamp = LocalDateTime.now();
        this.sequenceNumber = sequenceNumber;
        this.totalPackets = totalPackets;
        this.payload = payload;
        this.checksum = checksum;
        this.encryptionHash = encryptionHash;
        this.isAcknowledged = false;
    }

    // Getters
```

```
    public UUID getPacketId() { return packetId; }
    public String getSourceId() { return sourceId; }
    public String getDestinationId() { return destinationId; }
    public LocalDateTime getTimestamp() { return timestamp; }
    public int getSequenceNumber() { return sequenceNumber; }
    public int getTotalPackets() { return totalPackets; }
    public byte[] getPayload() { return payload; }
    public String getChecksum() { return checksum; }
    public String getEncryptionHash() { return encryptionHash; }
    public boolean isAcknowledged() { return isAcknowledged; }
    public void setAcknowledged(boolean acknowledged) { isAcknowledged = acknowledged; }

    public int getPayloadSize() {
        return payload != null ? payload.length : 0;
    }

    @Override
    public String toString() {
        return String.format(
            "DataPacket[ID: %s, From: %s, To: %s, Seq: %d/%d, Size: %d bytes, Time: %s]",
            packetId.toString().substring(0, 8), sourceId, destinationId,
            sequenceNumber, totalPackets, getPayloadSize(),
            timestamp.format(DateTimeFormatter.ofPattern("HH:mm:ss.SSS"))
        );
    }
}

// Acknowledgment packet
class AckPacket implements Serializable {
    private static final long serialVersionUID = 1L;

    private final UUID originalPacketId;
    private final String destinationId;
    private final LocalDateTime timestamp;
    private final boolean success;
    private final String message;
    private final int nextExpectedSequence;

    public AckPacket(UUID originalPacketId, String destinationId, boolean success,
                     String message, int nextExpectedSequence) {
        this.originalPacketId = originalPacketId;
        this.destinationId = destinationId;
        this.timestamp = LocalDateTime.now();
        this.success = success;
        this.message = message;
        this.nextExpectedSequence = nextExpectedSequence;
    }

    // Getters
    public UUID getOriginalPacketId() { return originalPacketId; }
    public String getDestinationId() { return destinationId; }
    public LocalDateTime getTimestamp() { return timestamp; }
    public boolean isSuccess() { return success; }
    public String getMessage() { return message; }
    public int getNextExpectedSequence() { return nextExpectedSequence; }

    @Override
    public String toString() {
        String status = success ? "SUCCESS" : "FAILED";
        return String.format("AckPacket[Original: %s, To: %s, Status: %s, Message: %s]",
                             originalPacketId.toString().substring(0, 8),
                             destinationId, status, message);
```

```
    }
}

// Data integrity and security utility
class SecurityUtils {
    private static final String HASH_ALGORITHM = "SHA-256";
    private static final String ENCRYPTION_ALGORITHM = "AES";
    private static final String SECRET_KEY = "DistributedSystemKey123"; // 24 chars for AES-192

    // Generate checksum for data integrity
    public static String generateChecksum(byte[] data) throws DistributedSystemException {
        try {
            MessageDigest digest = MessageDigest.getInstance(HASH_ALGORITHM);
            byte[] hash = digest.digest(data);
            return Base64.getEncoder().encodeToString(hash);
        } catch (NoSuchAlgorithmException e) {
            throw new DistributedSystemException("Checksum generation failed",
ErrorType.SECURITY_ERROR, e);
        }
    }

    // Verify checksum
    public static boolean verifyChecksum(byte[] data, String expectedChecksum) throws
DistributedSystemException {
        String actualChecksum = generateChecksum(data);
        return actualChecksum.equals(expectedChecksum);
    }

    // Encrypt data
    public static byte[] encrypt(byte[] data) throws DistributedSystemException {
        try {
            // In production, use proper key management
            byte[] key = Arrays.copyOf(SECRET_KEY.getBytes(), 24); // AES-192
            javax.crypto.spec.SecretKeySpec keySpec = new javax.crypto.spec.SecretKeySpec(key,
ENCRYPTION_ALGORITHM);
            javax.crypto.Cipher cipher = javax.crypto.Cipher.getInstance(ENCRYPTION_ALGORITHM);
            cipher.init(javax.crypto.Cipher.ENCRYPT_MODE, keySpec);
            return cipher.doFinal(data);
        } catch (Exception e) {
            throw new DistributedSystemException("Encryption failed", ErrorType.SECURITY_ERROR,
e);
        }
    }

    // Decrypt data
    public static byte[] decrypt(byte[] encryptedData) throws DistributedSystemException {
        try {
            byte[] key = Arrays.copyOf(SECRET_KEY.getBytes(), 24);
            javax.crypto.spec.SecretKeySpec keySpec = new javax.crypto.spec.SecretKeySpec(key,
ENCRYPTION_ALGORITHM);
            javax.crypto.Cipher cipher = javax.crypto.Cipher.getInstance(ENCRYPTION_ALGORITHM);
            cipher.init(javax.crypto.Cipher.DECRYPT_MODE, keySpec);
            return cipher.doFinal(encryptedData);
        } catch (Exception e) {
            throw new DistributedSystemException("Decryption failed", ErrorType.SECURITY_ERROR,
e);
        }
    }

    // Compress data
    public static byte[] compress(byte[] data) throws DistributedSystemException {
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
             DeflaterOutputStream dos = new DeflaterOutputStream(baos)) {
            dos.write(data);
            dos.finish();
            return baos.toByteArray();
        } catch (IOException e) {
            throw new DistributedSystemException("Compression failed",
ErrorType.SERIALIZATION_ERROR, e);
        }
    }

    // Decompress data
    public static byte[] decompress(byte[] compressedData) throws DistributedSystemException {
        try (ByteArrayInputStream bais = new ByteArrayInputStream(compressedData);
             InflaterInputStream iis = new InflaterInputStream(bais);
             ByteArrayOutputStream baos = new ByteArrayOutputStream()) {

            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = iis.read(buffer)) != -1) {
                baos.write(buffer, 0, bytesRead);
            }
            return baos.toByteArray();
        } catch (IOException e) {
            throw new DistributedSystemException("Decompression failed",
ErrorType.DESERIALIZATION_ERROR, e);
        }
    }
}

// Network communication utility
class NetworkUtils {

    // Serialize object to byte array
    public static byte[] serializeObject(Serializable obj) throws DistributedSystemException {
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
             ObjectOutputStream oos = new ObjectOutputStream(baos)) {
            oos.writeObject(obj);
            oos.flush();
            return baos.toByteArray();
        } catch (IOException e) {
            throw new DistributedSystemException("Serialization failed",
ErrorType.SERIALIZATION_ERROR, e);
        }
    }

    // Deserialize object from byte array
    public static Object deserializeObject(byte[] data) throws DistributedSystemException {
        try (ByteArrayInputStream bais = new ByteArrayInputStream(data);
             ObjectInputStream ois = new ObjectInputStream(bais)) {
            return ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
            throw new DistributedSystemException("Deserialization failed",
ErrorType.DESERIALIZATION_ERROR, e);
        }
    }

    // Send object over socket with retry mechanism
    public static void sendObject(Socket socket, Serializable obj, int maxRetries)
            throws DistributedSystemException {

        DistributedSystemException lastException = null;
```

```
        for (int attempt = 1; attempt <= maxRetries; attempt++) {
            try {
                OutputStream outputStream = socket.getOutputStream();
                ObjectOutputStream oos = new ObjectOutputStream(outputStream);
                oos.writeObject(obj);
                oos.flush();
                return; // Success

            } catch (IOException e) {
                lastException = new DistributedSystemException(
                    String.format("Send attempt %d/%d failed", attempt, maxRetries),
                    ErrorType.NETWORK_ERROR, e
                );

                System.err.println(lastException.getMessage());

                if (attempt < maxRetries) {
                    try {
                        Thread.sleep(1000 * attempt); // Exponential backoff
                    } catch (InterruptedException ie) {
                        Thread.currentThread().interrupt();
                        throw new DistributedSystemException("Send interrupted",
ErrorType.NETWORK_ERROR, ie);
                    }
                }
            }
        }

        throw lastException; // All retries failed
    }

    // Receive object from socket with timeout
    public static Object receiveObject(Socket socket, int timeoutMs) throws
DistributedSystemException {
        try {
            socket.setSoTimeout(timeoutMs);
            InputStream inputStream = socket.getInputStream();
            ObjectInputStream ois = new ObjectInputStream(inputStream);
            return ois.readObject();

        } catch (SocketTimeoutException e) {
            throw new DistributedSystemException("Receive timeout", ErrorType.TIMEOUT, e);
        } catch (IOException | ClassNotFoundException e) {
            throw new DistributedSystemException("Receive failed", ErrorType.NETWORK_ERROR, e);
        }
    }
}

// Distributed node base class
abstract class DistributedNode {
    protected final String nodeId;
    protected final int port;
    protected volatile boolean isRunning;
    protected final Map<UUID, DataPacket> sentPackets;
    protected final Map<UUID, DataPacket> receivedPackets;
    protected final ScheduledExecutorService scheduler;
    protected final Random random;

    public DistributedNode(String nodeId, int port) {
        this.nodeId = nodeId;
        this.port = port;
        this.isRunning = false;
```

```
        this.sentPackets = new ConcurrentHashMap<>();
        this.receivedPackets = new ConcurrentHashMap<>();
        this.scheduler = Executors.newScheduledThreadPool(2);
        this.random = new Random();
    }

    public String getNodeId() { return nodeId; }
    public int getPort() { return port; }
    public boolean isRunning() { return isRunning; }

    public abstract void start() throws DistributedSystemException;
    public abstract void stop() throws DistributedSystemException;

    // Simulate network issues for testing
    protected boolean shouldSimulateNetworkIssue() {
        return random.nextDouble() < 0.1; // 10% chance of network issue
    }

    protected void simulateNetworkDelay() {
        try {
            Thread.sleep(random.nextInt(100)); // 0-100ms delay
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

// Server node
class ServerNode extends DistributedNode {
    private ServerSocket serverSocket;
    private final ExecutorService clientHandlerPool;
    private final Map<String, List<DataPacket>> clientDataBuffers;

    public ServerNode(String nodeId, int port) {
        super(nodeId, port);
        this.clientHandlerPool = Executors.newCachedThreadPool();
        this.clientDataBuffers = new ConcurrentHashMap<>();
    }

    @Override
    public void start() throws DistributedSystemException {
        try {
            serverSocket = new ServerSocket(port);
            isRunning = true;

            System.out.println("ðŸš€ Server " + nodeId + " started on port " + port);

            // Start accepting connections
            new Thread(this::acceptConnections, "Server-Acceptor-" + nodeId).start();

            // Start packet processor
            scheduler.scheduleAtFixedRate(this::processBufferedPackets, 1, 1, TimeUnit.SECONDS);

        } catch (IOException e) {
            throw new DistributedSystemException("Failed to start server",
ErrorType.NETWORK_ERROR, e);
        }
    }

    @Override
    public void stop() throws DistributedSystemException {
        isRunning = false;
```

```
        try {
            if (serverSocket != null) {
                serverSocket.close();
            }
            clientHandlerPool.shutdown();
            scheduler.shutdown();
            System.out.println("ðŸ›‘ Server " + nodeId + " stopped");
        } catch (IOException e) {
            throw new DistributedSystemException("Error stopping server",
ErrorType.NETWORK_ERROR, e);
        }
    }

    private void acceptConnections() {
        while (isRunning) {
            try {
                Socket clientSocket = serverSocket.accept();
                System.out.println("ðŸ”— New client connected: " +
clientSocket.getInetAddress());

                clientHandlerPool.submit(() -> handleClient(clientSocket));

            } catch (IOException e) {
                if (isRunning) {
                    System.err.println("â Error accepting client connection: " +
e.getMessage());
                }
            }
        }
    }

    private void handleClient(Socket clientSocket) {
        String clientId = clientSocket.getInetAddress() + ":" + clientSocket.getPort();

        try {
            while (isRunning && !clientSocket.isClosed()) {
                // Receive data packet
                Object receivedObj = NetworkUtils.receiveObject(clientSocket, 5000);

                if (receivedObj instanceof DataPacket) {
                    DataPacket packet = (DataPacket) receivedObj;
                    handleDataPacket(packet, clientSocket);

                } else if (receivedObj instanceof AckPacket) {
                    AckPacket ack = (AckPacket) receivedObj;
                    handleAckPacket(ack);
                }
            }
        } catch (DistributedSystemException e) {
            System.err.println("â Error handling client " + clientId + ": " +
e.getMessage());
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                System.err.println("Error closing client socket: " + e.getMessage());
            }
        }
    }

    private void handleDataPacket(DataPacket packet, Socket clientSocket) throws
DistributedSystemException {
```

```
        System.out.println("ðŸ“¥ Server received: " + packet);

        // Verify checksum
        if (!SecurityUtils.verifyChecksum(packet.getPayload(), packet.getChecksum())) {
            System.err.println("â Checksum mismatch for packet: " + packet.getPacketId());
            sendAck(clientSocket, packet.getPacketId(), false, "Checksum mismatch",
                   packet.getSequenceNumber());
            return;
        }

        // Store packet in buffer
        clientDataBuffers
            .computeIfAbsent(packet.getSourceId(), k -> Collections.synchronizedList(new
ArrayList<>()))
            .add(packet);

        receivedPackets.put(packet.getPacketId(), packet);

        // Send acknowledgment
        sendAck(clientSocket, packet.getPacketId(), true, "Packet received successfully",
               packet.getSequenceNumber() + 1);

        System.out.println("âœ… Packet processed: " + packet.getPacketId());
    }

    private void handleAckPacket(AckPacket ack) {
        DataPacket originalPacket = sentPackets.get(ack.getOriginalPacketId());
        if (originalPacket != null) {
            originalPacket.setAcknowledged(ack.isSuccess());
            System.out.println("ðŸ“¨ Server received ACK for: " + originalPacket.getPacketId() +
                               " - " + (ack.isSuccess() ? "SUCCESS" : "FAILED"));
        }
    }

    private void sendAck(Socket clientSocket, UUID packetId, boolean success,
                         String message, int nextExpectedSeq) throws DistributedSystemException {
        AckPacket ack = new AckPacket(packetId, nodeId, success, message, nextExpectedSeq);
        NetworkUtils.sendObject(clientSocket, ack, 3);
        System.out.println("ðŸ“¤ Server sent ACK: " + ack);
    }

    private void processBufferedPackets() {
        for (Map.Entry<String, List<DataPacket>> entry : clientDataBuffers.entrySet()) {
            String clientId = entry.getKey();
            List<DataPacket> packets = entry.getValue();

            // Sort packets by sequence number
            packets.sort(Comparator.comparingInt(DataPacket::getSequenceNumber));

            // Process complete sequences
            List<DataPacket> completeSequence = new ArrayList<>();
            int expectedSeq = 0;

            for (DataPacket packet : packets) {
                if (packet.getSequenceNumber() == expectedSeq) {
                    completeSequence.add(packet);
                    expectedSeq++;
                } else {
                    break; // Missing packet in sequence
                }
            }
```

```
            if (!completeSequence.isEmpty() &&
                completeSequence.get(completeSequence.size() - 1).getSequenceNumber() + 1 ==
                completeSequence.get(completeSequence.size() - 1).getTotalPackets()) {

                // All packets received, process the complete data
                processCompleteData(clientId, completeSequence);

                // Remove processed packets
                packets.removeAll(completeSequence);
            }
        }
    }

    private void processCompleteData(String clientId, List<DataPacket> packets) {
        try {
            // Reconstruct original data
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            for (DataPacket packet : packets) {
                baos.write(packet.getPayload());
            }

            byte[] completeData = baos.toByteArray();

            // Decompress and decrypt
            byte[] decompressedData = SecurityUtils.decompress(completeData);
            byte[] decryptedData = SecurityUtils.decrypt(decompressedData);

            // Deserialize the object
            Object originalObject = NetworkUtils.deserializeObject(decryptedData);

            System.out.println("ðŸŽ‰ Server successfully reconstructed object from " +
                               packets.size() + " packets for client: " + clientId);
            System.out.println("   Original object: " + originalObject);

        } catch (DistributedSystemException e) {
            System.err.println("â Œ Error processing complete data from client " + clientId +
": " + e.getMessage());
        }
    }
}

// Client node
class ClientNode extends DistributedNode {
    private final String serverHost;
    private final int serverPort;
    private final Map<UUID, CompletableFuture<AckPacket>> pendingAcks;

    public ClientNode(String nodeId, String serverHost, int serverPort) {
        super(nodeId, serverPort + 1000 + new Random().nextInt(1000)); // Dynamic client port
        this.serverHost = serverHost;
        this.serverPort = serverPort;
        this.pendingAcks = new ConcurrentHashMap<>();
    }

    @Override
    public void start() throws DistributedSystemException {
        isRunning = true;

        // Start acknowledgment handler
        scheduler.scheduleAtFixedRate(this::checkPendingAcks, 1, 2, TimeUnit.SECONDS);

        System.out.println("ðŸš€ Client " + nodeId + " started, connecting to " + serverHost +
```

```
":" + serverPort);
    }

    @Override
    public void stop() throws DistributedSystemException {
        isRunning = false;
        scheduler.shutdown();
        System.out.println("ðŸ›‘ Client " + nodeId + " stopped");
    }

    public void sendData(Serializable data) throws DistributedSystemException {
        try {
            // Serialize and prepare data
            byte[] serializedData = NetworkUtils.serializeObject(data);
            byte[] encryptedData = SecurityUtils.encrypt(serializedData);
            byte[] compressedData = SecurityUtils.compress(encryptedData);

            // Split into packets (simulate large data transfer)
            List<byte[]> packetPayloads = splitData(compressedData, 1024); // 1KB chunks

            System.out.println("ðŸ“¦ Client splitting data into " + packetPayloads.size() + "
packets");

            // Send packets
            for (int i = 0; i < packetPayloads.size(); i++) {
                byte[] payload = packetPayloads.get(i);
                String checksum = SecurityUtils.generateChecksum(payload);
                String encryptionHash = SecurityUtils.generateChecksum(encryptedData);

                DataPacket packet = new DataPacket(nodeId, "SERVER", i,
                                                   packetPayloads.size(), payload, checksum,
encryptionHash);

                sendPacketWithRetry(packet, 3);

                // Small delay between packets
                Thread.sleep(50);
            }

        } catch (Exception e) {
            throw new DistributedSystemException("Failed to send data", ErrorType.NETWORK_ERROR,
e);
        }
    }

    private List<byte[]> splitData(byte[] data, int chunkSize) {
        List<byte[]> chunks = new ArrayList<>();
        for (int i = 0; i < data.length; i += chunkSize) {
            int end = Math.min(data.length, i + chunkSize);
            chunks.add(Arrays.copyOfRange(data, i, end));
        }
        return chunks;
    }

    private void sendPacketWithRetry(DataPacket packet, int maxRetries) throws
DistributedSystemException {
        for (int attempt = 1; attempt <= maxRetries; attempt++) {
            try (Socket socket = new Socket(serverHost, serverPort)) {

                if (shouldSimulateNetworkIssue()) {
                    throw new IOException("Simulated network issue");
                }
```

```
                simulateNetworkDelay();

                // Send packet
                NetworkUtils.sendObject(socket, packet, 2);
                sentPackets.put(packet.getPacketId(), packet);

                // Wait for acknowledgment
                CompletableFuture<AckPacket> ackFuture = new CompletableFuture<>();
                pendingAcks.put(packet.getPacketId(), ackFuture);

                // Receive acknowledgment with timeout
                Object response = NetworkUtils.receiveObject(socket, 5000);

                if (response instanceof AckPacket) {
                    AckPacket ack = (AckPacket) response;
                    ackFuture.complete(ack);
                    pendingAcks.remove(packet.getPacketId());

                    if (ack.isSuccess()) {
                        packet.setAcknowledged(true);
                        System.out.println("âœ… Packet " + packet.getSequenceNumber() + "
acknowledged");
                        return;
                    } else {
                        System.err.println("â Packet " + packet.getSequenceNumber() + "
failed: " + ack.getMessage());
                    }
                }

            } catch (DistributedSystemException e) {
                System.err.println("â Send attempt " + attempt + "/" + maxRetries + " failed:
" + e.getMessage());

                if (attempt == maxRetries) {
                    throw e;
                }

                try {
                    Thread.sleep(1000 * attempt); // Exponential backoff
                } catch (InterruptedException ie) {
                    Thread.currentThread().interrupt();
                    throw new DistributedSystemException("Send interrupted",
ErrorType.NETWORK_ERROR, ie);
                }
            } catch (IOException e) {
                System.err.println("â Connection error on attempt " + attempt + "/" +
maxRetries + ": " + e.getMessage());

                if (attempt == maxRetries) {
                    throw new DistributedSystemException("All send attempts failed",
ErrorType.CONNECTION_LOST, e);
                }

                try {
                    Thread.sleep(1000 * attempt);
                } catch (InterruptedException ie) {
                    Thread.currentThread().interrupt();
                    throw new DistributedSystemException("Send interrupted",
ErrorType.NETWORK_ERROR, ie);
                }
            }
```

```
        }
    }

    private void checkPendingAcks() {
        Iterator<Map.Entry<UUID, CompletableFuture<AckPacket>>> iterator =
pendingAcks.entrySet().iterator();

        while (iterator.hasNext()) {
            Map.Entry<UUID, CompletableFuture<AckPacket>> entry = iterator.next();
            UUID packetId = entry.getKey();
            CompletableFuture<AckPacket> future = entry.getValue();

            DataPacket packet = sentPackets.get(packetId);
            if (packet != null &&
packet.getTimestamp().plusSeconds(30).isBefore(LocalDateTime.now())) {
                // Packet timeout
                future.completeExceptionally(new DistributedSystemException("Packet timeout",
ErrorType.TIMEOUT));
                iterator.remove();
                System.err.println("â° Packet timeout: " + packetId);
            }
        }
    }
}

// Sample data class for transmission
class SensorData implements Serializable {
    private static final long serialVersionUID = 1L;

    private final String sensorId;
    private final double temperature;
    private final double humidity;
    private final double pressure;
    private final LocalDateTime timestamp;

    public SensorData(String sensorId, double temperature, double humidity, double pressure) {
        this.sensorId = sensorId;
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        this.timestamp = LocalDateTime.now();
    }

    // Getters
    public String getSensorId() { return sensorId; }
    public double getTemperature() { return temperature; }
    public double getHumidity() { return humidity; }
    public double getPressure() { return pressure; }
    public LocalDateTime getTimestamp() { return timestamp; }

    @Override
    public String toString() {
        return String.format("SensorData[ID: %s, Temp: %.2fÂ°C, Humidity: %.1f%%, Pressure:
%.1fhPa, Time: %s]",
                             sensorId, temperature, humidity, pressure,
                             timestamp.format(DateTimeFormatter.ofPattern("HH:mm:ss")));
    }
}

class SystemMetrics implements Serializable {
    private static final long serialVersionUID = 1L;
```

```
    private final String nodeId;
    private final double cpuUsage;
    private final double memoryUsage;
    private final long freeDiskSpace;
    private final int activeConnections;
    private final LocalDateTime timestamp;

    public SystemMetrics(String nodeId, double cpuUsage, double memoryUsage,
                         long freeDiskSpace, int activeConnections) {
        this.nodeId = nodeId;
        this.cpuUsage = cpuUsage;
        this.memoryUsage = memoryUsage;
        this.freeDiskSpace = freeDiskSpace;
        this.activeConnections = activeConnections;
        this.timestamp = LocalDateTime.now();
    }

    @Override
    public String toString() {
        return String.format("SystemMetrics[Node: %s, CPU: %.1f%%, Memory: %.1f%%, Disk: %dMB,
Connections: %d]",
                             nodeId, cpuUsage, memoryUsage, freeDiskSpace / (1024 * 1024),
activeConnections);
    }
}

// Distributed System Manager
public class DistributedSystemManager {
    private final List<DistributedNode> nodes;
    private final ExecutorService executor;

    public DistributedSystemManager() {
        this.nodes = new ArrayList<>();
        this.executor = Executors.newCachedThreadPool();
    }

    public void addNode(DistributedNode node) {
        nodes.add(node);
    }

    public void startSystem() {
        System.out.println("ðŸš€ Starting Distributed System...");

        for (DistributedNode node : nodes) {
            executor.submit(() -> {
                try {
                    node.start();
                } catch (DistributedSystemException e) {
                    System.err.println("â Œ Failed to start node " + node.getNodeId() + ": " +
e.getMessage());
                }
            });
        }
    }

    public void stopSystem() {
        System.out.println("ðŸ›‘ Stopping Distributed System...");

        for (DistributedNode node : nodes) {
            try {
                node.stop();
            } catch (DistributedSystemException e) {
```

```
                System.err.println("â Error stopping node " + node.getNodeId() + ": " +
e.getMessage());
            }
        }

        executor.shutdown();
    }

    public static void main(String[] args) {
        DistributedSystemManager system = new DistributedSystemManager();

        // Create server node
        ServerNode server = new ServerNode("SERVER-1", 8080);
        system.addNode(server);

        // Create client nodes
        ClientNode client1 = new ClientNode("CLIENT-1", "localhost", 8080);
        ClientNode client2 = new ClientNode("CLIENT-2", "localhost", 8080);
        ClientNode client3 = new ClientNode("CLIENT-3", "localhost", 8080);

        system.addNode(client1);
        system.addNode(client2);
        system.addNode(client3);

        // Start the system
        system.startSystem();

        // Wait a bit for servers to start
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Send some test data from clients
        executorSubmit(system.executor, () -> {
            try {
                client1.start();
                sendTestData(client1, "Temperature-Sensor-1");
            } catch (DistributedSystemException e) {
                System.err.println("Client 1 error: " + e.getMessage());
            }
        });

        executorSubmit(system.executor, () -> {
            try {
                client2.start();
                sendTestData(client2, "Humidity-Sensor-2");
            } catch (DistributedSystemException e) {
                System.err.println("Client 2 error: " + e.getMessage());
            }
        });

        executorSubmit(system.executor, () -> {
            try {
                client3.start();
                sendTestData(client3, "Pressure-Sensor-3");
            } catch (DistributedSystemException e) {
                System.err.println("Client 3 error: " + e.getMessage());
            }
        });
```

```
        // Let the system run for a while
        try {
            Thread.sleep(30000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Stop the system
        system.stopSystem();
    }

    private static void executorSubmit(ExecutorService executor, Runnable task) {
        try {
            executor.submit(task);
        } catch (Exception e) {
            System.err.println("Error submitting task: " + e.getMessage());
        }
    }

    private static void sendTestData(ClientNode client, String sensorId) {
        Random random = new Random();

        for (int i = 0; i < 3; i++) {
            try {
                // Create sample data
                SensorData sensorData = new SensorData(
                    sensorId,
                    20 + random.nextDouble() * 10, // 20-30Â°C
                    40 + random.nextDouble() * 30, // 40-70%
                    1000 + random.nextDouble() * 100 // 1000-1100 hPa
                );

                SystemMetrics metrics = new SystemMetrics(
                    client.getNodeId(),
                    random.nextDouble() * 100,
                    random.nextDouble() * 100,
                    random.nextLong() % 1000000000,
                    random.nextInt(10)
                );

                // Send data
                System.out.println("\nðŸ“¤ " + client.getNodeId() + " sending data batch " + (i
+ 1));
                client.sendData(sensorData);
                Thread.sleep(1000);
                client.sendData(metrics);

                // Wait before next batch
                Thread.sleep(5000);

            } catch (Exception e) {
                System.err.println("â Error in test data sending: " + e.getMessage());
            }
        }
    }
}
```