Módulo 1: Connection Manager (Telefonista)

Clase afectada: P2PConnectionManager.java

1. Extraer la lógica de inicialización del NetServer del método start() a un método privado independiente initServer(int port).
2. Extraer la lógica del cliente a un método connectToPeer(String host, int port) que permita reconexiones dinámicas.
3. Estandarizar el "sobre" JSON de red. Todo byte recibido debe ser parseable a una estructura { "type": "...", "payload": {...} }.
4. Modificar el RecordParser para incluir un límite máximo de bytes en la cabecera de longitud, previniendo ataques de agotamiento de memoria (OOM).
5. Envolver la deserialización (de Buffer a JsonObject) en un try-catch estricto; si el JSON está malformado, cerrar el NetSocket inmediatamente.
6. Crear un método unificado sendBytes(NetSocket socket, JsonObject data) que centralice la creación del *Framing* (Longitud + Payload).
7. Añadir opciones de Timeout al NetClient para que las conexiones a nodos caídos no se queden colgadas indefinidamente.
8. Implementar el método stop() del Verticle para garantizar el cierre limpio (graceful shutdown) de todos los sockets activos al detener el nodo.
9. Definir una dirección de EventBus puramente interna (ej. p2p.internal.raw) para despachar los JSONs limpios, aislando a la red de la lógica de negocio.
10. Pactar con el Módulo 2 la estructura exacta del campo type del JSON para que ellos sepan cómo enrutar el mensaje.

Módulo 2: P2P Discovery & Handshake (Relaciones Públicas)

Clase afectada: P2PConnectionManager.java (Debe extraerse a PeerManager.java)

1. Mover la lista activeSockets a un nuevo componente o Verticle dedicado exclusivamente a la topología de red.
2. Sustituir la lista simple por un ConcurrentHashMap<String, NetSocket> que mapee identificadores de nodo (IP:Puerto) con sus sockets.
3. Reescribir el proceso de Handshake actual: debe ser bidireccional (Request/Response) e incluir la validación de la VERSION del software.
4. Implementar un mecanismo periódico (vertx.setPeriodic) que envíe mensajes de tipo PING a todos los peers del mapa.
5. Añadir lógica de expulsión: si un peer no responde con un PONG en X segundos, eliminarlo del ConcurrentHashMap y cerrar su socket.
6. Modificar la función broadcastMessageExcept para iterar sobre el nuevo mapa seguro, garantizando que el origen del mensaje nunca reciba el rebote.
7. Implementar el tipo de mensaje GET_PEERS: al conectar, un nodo debe pedirle a su semilla la lista de IPs que conoce.
8. Procesar el mensaje de respuesta PEER_LIST para iniciar conexiones NetClient automáticas a los nuevos nodos descubiertos (Gossip).
9. Publicar eventos en el EventBus (network.peer.connected, network.peer.disconnected) para que otros módulos puedan monitorizar la salud de la red.

Módulo 3: Blockchain Storage (Bibliotecario)

Clase afectada: BlockChain.java

1. Aislar esta clase. Ya no debe contener la lógica de validación de Proof of Work ni verificación de hashes al añadir un bloque.
2. Integrar el FileSystem de Vert.x para guardar el estado. Definir una ruta estática (ej. data/blockchain.json). Es imperativo utilizar la API asíncrona proporcionada por el framework (vertx.fileSystem().writeFile(...)) en lugar de las clases síncronas tradicionales de Java (java.io.File). Esto garantiza que las operaciones de entrada/salida (I/O) al disco duro no bloqueen el EventLoop del nodo.
3. Modificar addBlock() para que, tras añadir a la memoria, serialice el bloque y haga un append seguro al archivo en disco.
4. Crear un método loadChainFromDisk() que se ejecute en el constructor para reconstruir la List<Block> si el nodo se reinició.
5. Refactorizar createGenesisBlock() para que solo se ejecute si loadChainFromDisk() detecta que el archivo no existe o está vacío.
6. Sustituir las búsquedas secuenciales por índices. Mantener un HashMap<String, Block> en memoria para búsquedas $O(1)$ por Hash.
7. Crear el método getBlockByIndex(long index) para soportar la futura sincronización entre nodos.
8. Implementar validación de integridad al leer del disco: si un bloque cargado desde archivo tiene el hash corrupto, detener el arranque del nodo.
9. Crear un método getBlocksFromIndex(long startIndex) que devuelva una sublista de bloques para responder a peticiones de nodos atrasados.

Módulo 4: Block Validator (Agente de Aduanas)

Clase afectada: Nueva clase BlockValidator.java

1. Extraer toda la lógica de los if de validación que actualmente ensucian el método addBlock de BlockChain.java.
2. Crear el método validatePoW(Block block) que recalcule independientemente el hash y verifique los ceros exigidos por la dificultad.
3. Crear validateContinuity(Block block, Block lastBlock) para asegurar que el previousHash enlaza matemáticamente y el index es secuencial.
4. Implementar comprobación de tiempo: rechazar el bloque si su timestamp es menor que el del bloque anterior o está más de 2 horas en el futuro.
5. Crear validateMerkleRoot(Block block) para forzar el recálculo del árbol desde el Body y compararlo con el Header.
6. Implementar la lógica de "Orphan Blocks": si un bloque es válido pero su padre no existe localmente, guardarlo temporalmente en memoria.
7. Verificar que el primer elemento del Body es estrictamente una CoinbaseTransaction y que no hay ninguna otra en el resto del bloque.
8. Establecer la coordinación con el Módulo 5: iterar sobre las transacciones del bloque entrante delegando la verificación de firmas al notario.
9. Suscribirse al EventBus en INCOMING_BLOCK. Solo si pasa los 8 pasos anteriores, invocar la escritura en el Módulo 3.
10. Definir una política estricta de rechazo: al primer fallo en la validación, abortar el procesamiento del bloque y registrar un warning en consola.

Módulo 5: Transaction Validator (Notario)

Clases afectadas: Transaction.java, nueva clase TransactionValidator.java

1. Extraer el método verifySignature() del interior de Transaction.java. Las entidades deben ser anémicas; la lógica va en el servicio validador.
2. Desarrollar la estructura de datos para el Estado (Balance o UTXO Set) en memoria, separada del almacenamiento de bloques.
3. Crear el método checkFunds(Transaction tx) que consulte el estado local para certificar que el emisor tiene saldo >= amount.
4. Implementar reglas de formato: rechazar transacciones si amount <= 0 o si sender.equals(receiver).
5. Verificar matemáticamente el ID: recalcular el hash SHA-256 de los datos originales y comparar si coincide con el transactionId declarado.
6. Reescribir la comprobación ECDSA (SecurityUtils.verifyECDSASig) dentro de un método aislado validateAuthenticity(Transaction tx).
7. Crear un método updateState(Block block) que, una vez confirmado un bloque, sume y reste saldos en la estructura de Estado.
8. Generar una variante del validador para la Mempool (validateForMempool) que deduzca los saldos basándose en operaciones pendientes, no solo confirmadas.
9. Implementar comprobación anti-replay: verificar la existencia del Nonce o consumo estricto de UTXO.
10. Coordinar con el Módulo 4 para que rechace bloques enteros si una sola transacción contenida falla esta auditoría.

Módulo 6: Mempool (Sala de Espera)

Clase afectada: MinerVerticle.java (Debe extraerse a MempoolManager.java)

1. Sacar la lista transactionPool fuera del Verticle del minero, dándole su propio ciclo de vida.
2. Reemplazar el ArrayList por una PriorityQueue (Cola de Prioridad), ordenada temporalmente (o por fee si lo implementáis).
3. Añadir un ConcurrentHashMap en paralelo a la cola para búsqueda $O(1)$, evitando iterar para rechazar duplicados (idempotencia).
4. Crear el método pullTransactions(int limit) que extraiga de la cola los elementos con mayor prioridad sin borrarlos definitivamente.
5. Implementar purgeConfirmed(List<Transaction> minedTxs): cuando el Módulo 4 acepta un bloque, la Mempool debe borrar esas TXs de su cola.
6. Definir un límite máximo de memoria (ej. tamaño máximo = 5000 TXs). Implementar política de expulsión de las más antiguas si se llena.
7. Escuchar el EventBus en INCOMING_TRANSACTION o NEW_TRANSACTION, delegando en el Módulo 5 (Validator) antes de encolar.
8. Añadir lógica de caducidad: revisar periódicamente la cola y eliminar transacciones que lleven esperando más de X horas sin ser minadas.
9. Crear un endpoint del EventBus mempool.status que devuelva el tamaño actual, útil para el Módulo 7.
10. Establecer el umbral de disparo: enviar un evento (ej. mempool.ready) al Módulo 7 únicamente cuando se alcance el BLOCK_SIZE pactado.

Módulo 7: Miner CPU (Obrero)

Clase afectada: MinerVerticle.java

1. Limpiar el Verticle aislando el bucle matemático en una clase worker independiente PoWMiner.java.
2. Sustituir la variable boolean isMining por un AtomicBoolean para garantizar acceso concurrente seguro (Thread-Safe) desde múltiples hilos.
3. Modificar el bucle while (!hash.startsWith(target)) para incluir una condición de salida de emergencia basada en el AtomicBoolean.
4. Suscribirse a INCOMING_BLOCK: si un nuevo bloque válido es añadido a la cadena (vía Módulo 4), disparar el AtomicBoolean a falso para cancelar el trabajo inútil.
5. Implementar la generación programática de la CoinbaseTransaction garantizando que sea la TX con índice 0 del nuevo bloque.
6. Eliminar la recolección de variables estáticas. El minero debe solicitar dinámicamente el previousHash y difficulty al Módulo 3 antes de arrancar.
7. Introducir actualización temporal: dentro del bucle de minado, actualizar el timestamp del Header periódicamente para refrescar el hash a intentar.
8. Gestionar el *Nonce Overflow*: si el nonce alcanza Long.MAX_VALUE, atraparlo, resetear a 0 y forzar la actualización del timestamp.
9. Aislar la invocación a vertx.executeBlocking controlando los errores del callback de manera que el hilo no muera si la red se desincroniza.
10. Exigir al Módulo 6 que reintegre las transacciones a la cola si el minado es interrumpido antes de completarse.

Módulo 8: Wallet Keys (Cerrajero)

Clases afectadas: Wallet.java, SecurityUtils.java

1. **Imprescindible:** Eliminar referencias a RSA si las hay. Mantener generación y firma basada estrictamente en ECDSA (secp256r1).
2. Extraer la lógica de inicialización criptográfica a un KeyManager.java que controle el ciclo de vida del par de claves.
3. Implementar I/O en disco: al crear la Wallet, serializar la PrivateKey y guardarla en un archivo local (wallet.dat).
4. Implementar carga desde disco: al arrancar el nodo, si existe wallet.dat, reconstruir las claves en lugar de generar unas nuevas aleatorias.
5. Añadir seguridad en reposo: cifrar el archivo wallet.dat usando AES, exigiendo una contraseña en la inicialización (vía variables de entorno o config).
6. Refactorizar la generación del Address. Aplicar Hash160 (SHA-256 seguido de compresión a 20 bytes/Base58) a la clave pública en lugar de usar Base64 crudo.
7. Aislar la firma: Crear un método encapsulado byte[] sign(byte[] dataHash) que actúe como caja negra; ningún otro módulo debe poder acceder al objeto PrivateKey.
8. Coordinar la interfaz estrictamente con el Módulo 9: el Módulo 8 provee la firma abstracta, pero no construye la transacción JSON.

Módulo 9: Transaction Builder (Gestor)

Clase afectada: WalletVerticle.java, Transaction.java

1. Diseñar la clase TransactionBuilder implementando el patrón Builder (new TransactionBuilder().from().to().amount().build()).
2. Desacoplar la estructura JSON de la lógica de red. El constructor asume parámetros crudos e invoca al KeyManager (Módulo 8) para inyectar la firma.
3. Garantizar el orden criptográfico: Calcular el hash de metadatos calculateHash() -> Firmar Hash -> Asignar firma al objeto final.
4. Añadir el campo fee a la clase Transaction, permitiendo calcular comisiones para el minero (si el modelo contable lo exige).
5. Implementar validación preventiva: antes de firmar, el Builder debe comprobar estáticamente que la cantidad no es negativa o nula.
6. Integrar lógica de UXTO "Change": si se gasta un output de 10 para enviar 2, el Builder debe autogenerar una salida de 8 de vuelta al emisor.
7. Encapsular la serialización: el objeto final debe exponer un método .toNetworkJson() estandarizado para el EventBus.
8. Sustituir el UUID aleatorio por el hash criptográfico real (transactionId) en las comprobaciones anti-bucles del payload.
9. Sustituir la lógica del generador simulado en WalletVerticle por invocaciones ordenadas a este Builder.
10. Pactar el canal de comunicación final: El Builder emite a una sola dirección de EventBus, y confía en el Módulo 6 para aceptarla o rechazarla.