Segunda entrega sistemas distribuidos

Cliente BitTorrent con Overlay Gossip

Jabel Resendiz Aguirre
Noel Pérez Calvo

Índice

1. Arquitectura

Esta sección describe la arquitectura general del sistema BitTorrent distribuido, incluyendo su organización, roles de los componentes, integración con Docker Swarm, patrones arquitecturales aplicados, y las decisiones de diseño que permiten operación descentralizada y escalable.

1.1. Organización de su sistema distribuido

La arquitectura del sistema evoluciona de un modelo cliente-servidor centralizado, donde un tracker único coordina a los pares, a un sistema distribuido sin un punto central de fallo. Cada nodo en la red es un igual (peer) y participa tanto en la descarga/subida de archivos como en el descubrimiento de otros nodos.

La organización se basa en una red superpuesta (overlay network) de tipo no estructurado. En esta red, cada nodo mantiene una lista de proveedores (peers que poseen un archivo) para cada 'infoHash' (identificador único del torrent). La información se propaga a través de un protocolo de gossip, eliminando la dependencia de un servidor central.

1.2. Roles de su sistema

En nuestro diseño, los roles son homogéneos, donde cada nodo del sistema desempeña simultáneamente múltiples responsabilidades de manera integrada. Como **Cliente BitTorrent**, cada nodo participa activamente en el enjambre (swarm) para descargar y compartir piezas de archivos utilizando el protocolo `peerwire`, gestionando tanto conexiones entrantes como salientes con otros peers. Simultáneamente, actúa como **Nodo de Overlay**, ejecutando el servicio de descubrimiento distribuido donde escucha peticiones de otros nodos, responde a búsquedas (`Lookup`) y propaga la información que conoce mediante gossip, mientras anuncia su propia presencia en la red para los torrents que está compartiendo.

No existen nodos con funciones especiales; la red es completamente plana y todos los participantes son iguales, lo que garantiza la descentralización total del sistema y elimina puntos únicos de fallo que podrían comprometer la disponibilidad global.

1.3. Integración con Docker Swarm

La arquitectura aprovecha las capacidades nativas de Docker Swarm para crear una red distribuida resiliente donde cada instancia de la aplicación se ejecuta como un servicio independiente con capacidad de auto-discovery. El modelo de despliegue se basa en servicios homogéneos donde todos los nodos ejecutan el mismo código con roles idénticos, eliminando puntos centralizados de fallo, mientras Docker Swarm permite aumentar o disminuir réplicas dinámicamente según la demanda. El sistema incorpora monitoreo automático de salud de contenedores con restart policies configurables y distribución automática de conexiones entrantes entre réplicas disponibles.

La red overlay de Docker proporciona comunicación transparente entre contenedores ubicados en diferentes máquinas físicas, implementando encriptación IPSEC automática para tráfico inter-nodo, DNS interno para resolución de nombres de servicios, y enrutamiento inteligente con load balancing incorporado. Cada servicio expone dos puertos principales con propósitos diferenciados: el puerto peerwire para conexiones entrantes del protocolo BitTorrent enfocadas en transferencia de piezas de archivos, y el puerto overlay

para el listener del gossip protocol dedicado al descubrimiento distribuido y manejo de metadatos.

El sistema utiliza un mecanismo de bootstrap híbrido que combina el flag `--bootstrap` con lista de peers conocidos para arranque inicial, integración con DNS interno de Swarm para descobrir otros servicios, expansion dinámica donde una vez conectado el overlay gossip permite descubrir peers adicionales transitivamente, y resilience mediante tolerancia a fallos de nodos bootstrap a través de múltiples puntos de entrada.

1.4. Patrones Arquitecturales y Principios de Diseño

El sistema implementa el patrón de red superpuesta (overlay network) que proporciona abstracción de la red física, permitiendo que los nodos se comuniquen a través de una topología lógica completamente independiente de la infraestructura subyacente. Esta abstracción habilita enrutamiento especializado mediante algoritmos de descubrimiento optimizados para contenido específico identificado por infoHash, mientras mantiene adaptabilidad topológica donde la red se reconfigura dinámicamente ante fallos o cambios de membresía sin intervención manual.

Los principios de diseño distribuido se fundamentan en descentralización total que elimina puntos únicos de fallo mediante replicación de información, consistencia eventual usando timestamps LWW (Last-Writer-Wins) para convergencia sin sincronización fuerte, y un enfoque CAP que privilegia disponibilidad y tolerancia a particiones sobre consistencia fuerte. El sistema incorpora escalabilidad horizontal con capacidad de agregar nodos sin reconfiguración central, complementado con mecanismos de anti-entropía que realizan reconciliación periódica para mantener coherencia del estado distribuido.

La arquitectura mantiene clara separación de responsabilidades organizando el sistema en múltiples planos especializados: el plano de control utiliza overlay gossip para descubrimiento y manejo de metadatos, el plano de datos emplea peerwire para transferencia eficiente de contenido, el layer de persistencia opera independientemente con validación criptográfica, y el sistema de configuración proporciona flexibilidad de deployment a través de flags y environment variables.

1.5. Modelo de Datos y Escalabilidad

La arquitectura define estructuras de datos especializadas optimizadas para diferentes responsabilidades del sistema distribuido. La estructura **ProviderMeta** encapsula metadatos esenciales de peers incluyendo dirección de red (Addr), identificador único (PeerID), estado de descarga (Left bytes), y timestamp de última actividad (LastSeen) crucial para algoritmos de anti-entropía. El **Store distribuido** implementa una estructura jerárquica `map[infoHash]map[addr]ProviderMeta` con protección `sync.RWMutex` que garantiza consultas concurrentes seguras y actualizaciones atómicas, mientras el **Bitfield comprimido** proporciona representación eficiente de piezas completadas usando 1 bit por pieza en lugar de arrays de booleanos, reduciendo el uso de memoria significativamente. La comunicación inter-nodo se estandariza mediante **Mensajes Wire** que utilizan protocolo JSON estructurado con campos `type`, `info_hash`, `providers`, y `limit`.

Para manejar el crecimiento del sistema, la arquitectura incorpora múltiples mecanismos de escalabilidad coordinados. La expiración automática de metadatos obsoletos mediante TTL adaptativo (90 segundos por defecto) previene crecimiento ilimitado del store, mientras el parámetro `limit` en operaciones lookup previene respuestas excesiva-

mente grandes que podrían saturar la red. El sistema implementa gossip selectivo donde el full-push se limita a peers bootstrap para reducir tráfico de red exponencial, complementado con paralelismo controlado que restringe el fanout a 3 peers máximo en discovery BFS evitando explosión combinatoria de consultas. Las optimizaciones de conectividad incluyen reutilización de conexiones TCP y conexiones bidireccionales que reducen substancialmente el overhead de handshakes repetitivos.

1.6. Topología de Red y Comunicación

El sistema implementa una arquitectura híbrida de dos redes superpuestas que operan simultáneamente con propósitos complementarios. La red de descubrimiento (Overlay Gossip) constituye una red no estructurada donde cada nodo mantiene conexiones con un subconjunto de peers bootstrap, estableciendo conectividad inicial a través de 2-5 peers especificados via `--bootstrap`. Esta red utiliza protocolo TCP con mensajes JSON estructurados para operaciones de gossip, announce y lookup, implementando un patrón de propagación epidemic gossip con intervalos de 8 segundos y detección de fallos mediante health checks periódicos cada 10 segundos con timeout de 800ms.

Paralelamente, la red de transferencia (BitTorrent Swarm) opera como una red dinámica peer-to-peer dedicada exclusivamente a transferencia real de datos. Los peers se descubren mediante lookup en el overlay gossip en lugar del tradicional tracker HTTP, estableciendo conexiones directas utilizando el protocolo BitTorrent peerwire estándar sobre TCP. El sistema mantiene un pool dinámico de conexiones peer con validación de handshake rigurosa, implementando estados de control de flujo choke/unchoke e interested/not-interested para optimizar la utilización del ancho de banda disponible.

2. Procesos

La implementación del sistema distribuido se fundamenta en un modelo de procesos concurrentes que maximiza la eficiencia y escalabilidad mediante el uso intensivo de goroutines de Go. Esta arquitectura permite manejar simultáneamente miles de conexiones y operaciones de red mientras mantiene un consumo de recursos optimizado.

2.1. Tipos de procesos dentro del sistema

El sistema se organiza como un único proceso del sistema operativo que aloja múltiples componentes concurrentes implementados como **goroutines** en Go, donde cada instancia ejecuta procesos lógicos especializados que operan de manera coordinada. El **proceso principal del cliente**, implementado en `main.go`, actúa como goroutine principal que orquesta el flujo de ejecución completo, gestionando el ciclo de vida del torrent, manejando señales del sistema SIGINT/SIGTERM a través del canal `sigChan`, coordinando el proceso de cierre limpio mediante `shutdownChan`, y supervisando la finalización de descarga con `completedChan`. Paralelamente, el gestor de configuración y estado procesa flags de línea de comandos como `--discovery-mode`, `--bootstrap`, y `--overlay-port`, carga metadatos del torrent, configura el almacenamiento en disco y establece la función `computeLeft()` para calcular bytes restantes.

El **subsistema de overlay** opera mediante múltiples goroutines especializadas que mantienen la red de descubrimiento distribuido. El listener TCP del overlay, implementado en `serveListener`, constituye una goroutine dedicada que acepta conexiones TCP

entrantes en el puerto especificado por `--overlay-port` (por defecto 6000), utilizando un bucle infinito con `ln.Accept()` y delegando cada conexión a una goroutine independiente mediante `handleConn()`. Cada procesador de mensajes del overlay maneja una goroutine por conexión entrante que decodifica mensajes JSON y los procesa según su tipo, donde los mensajes `gossip` y `announce` actualizan el store local, mientras los `lookup` responden con proveedores locales disponibles.

La propagación de información se gestiona mediante **gossip periódico**, una goroutine con ticker de 8 segundos que propaga el estado local del overlay a todos los peers bootstrap, implementando un mecanismo de anti-entropía que envía la lista completa de proveedores por `infoHash` a los nodos conocidos. El sistema incorpora **health check periódico** mediante una goroutine con ticker de 10 segundos que verifica la conectividad de los peers bootstrap usando conexiones TCP de prueba con timeout de 800ms, eliminando automáticamente nodos no responsivos de la lista activa. El proceso de discovery implementa un algoritmo BFS (Breadth-First Search) con TTL para explorar la red de overlay, consultando peers iniciales y expandiendo la búsqueda a través de lookups paralelos, construyendo efectivamente un grafo de nodos alcanzables para un `infoHash` específico.

El **subsistema BitTorrent (Peerwire)** gestiona la transferencia real de datos mediante componentes altamente especializados que operan concurrentemente. El **Manager de Conexiones Peer**, implementado en la clase `Manager`, coordina el estado global de las conexiones peer-to-peer manteniendo referencias a todas las conexiones activas y orquestando la descarga paralela de piezas mediante el método `DownloadPieceParallel()`, optimizando la utilización del ancho de banda disponible. El listener de conexiones entrantes, ejecutado por `StartListeningForIncomingPeers`, opera como una goroutine que acepta conexiones TCP entrantes de otros peers BitTorrent, valida rigurosamente el handshake verificando protocolo, `info_hash`, y `peer_id`, estableciendo conexiones bidireccionales seguras para transferencia de piezas.

Cada conexión peer establecida genera un procesador dedicado implementado en `ReadLoop`, una goroutine que lee continuamente mensajes del protocolo BitTorrent incluyendo choke, unchoke, interested, have, piece, y request, actualizando dinámicamente el estado de la conexión correspondiente para mantener sincronización perfecta. La selección inteligente de piezas se maneja mediante el **Piece Picker** (`NextPieceFor`), un algoritmo sofisticado que determina qué pieza descargar de cada peer basándose en disponibilidad local y remota, implementando inicialmente una estrategia "first-neededçon capacidad de evolucionar hacia rarest-first"para optimizar la distribución del contenido en el swarm.

El almacenamiento persistente se gestiona a través del **Gestor de Almacenamiento** (`DiskPieceStore`), un proceso crítico que maneja toda la escritura y lectura de piezas en disco, verifica meticulosamente hashes SHA-1 de piezas completadas para garantizar integridad, mantiene el bitfield local actualizado, y ejecuta callbacks de finalización mediante `OnPieceComplete()` para coordinar acciones posteriores a la completación de cada pieza.

Los procesos de eventos y notificaciones coordinan la comunicación del estado del nodo con el resto de la red mediante mecanismos asíncronos sofisticados. La **rutina de announces periódicos**, implementada en `StartPeriodicAnnounceRoutineOverlay`, opera como una goroutine con ticker basado en `trackerInterval` que envía announces regulares al overlay comunicando el estado actual mediante `computeLeft()`, manteniendo así la presencia activa del nodo en la red y permitiendo que otros peers conozcan su disponibilidad y progreso de descarga.

La **rutina de completion** (`StartCompletionAnnounceRoutineOverlay`) funciona

como una goroutine especializada que escucha continuamente el canal `completedChan`, detectando automáticamente cuando la descarga finaliza y enviando inmediatamente un announce de çompleted"que transforma el nodo de leecher a seeder completo, notificando a toda la red su nueva capacidad de servir el archivo completo. El gestor de señales del sistema constituye una goroutine crítica que captura señales SIGINT/SIGTERM del sistema operativo y coordina el cierre ordenado de todos los subsistemas, asegurando que se envíen announces de "stopped.ªntes de terminar, garantizando así un shutdown limpio que permite a otros nodos actualizar sus listas de peers disponibles.

2.2. Organización y agrupación de los procesos

La coordinación entre los múltiples procesos concurrentes requiere una organización jerárquica bien definida que optimice la comunicación y minimice los conflictos de recursos.

La arquitectura agrupa los procesos en **tres subsistemas principales** que operan de forma concurrente pero coordinada, donde cada subsistema tiene responsabilidades específicas y comparte recursos de manera controlada. El subsistema de Overlay Gossip agrupa todos los procesos relacionados con el descubrimiento distribuido, incluyendo listener TCP, procesamiento de mensajes, gossip periódico, health checks, y discovery, compartiendo el objeto `Store` que mantiene el mapeo `infoHash ->providers` protegido por mutex `sync.RWMutex` para garantizar acceso concurrente seguro.

El **subsistema BitTorrent/Peerwire** incluye el manager de conexiones, listeners de peers entrantes, loops de lectura por conexión, piece picker, y almacenamiento en disco, operando sobre el objeto `DiskPieceStore` que gestiona tanto el estado de descarga como el bitfield local, coordinando todas las operaciones de transferencia de datos. El **subsistema de Control y Eventos** comprende el proceso principal, gestores de configuración, rutinas de announces periódicos, handlers de completion, y captura de señales, orquestando la interacción entre el overlay y el peerwire para mantener coherencia del sistema completo.

Todos los subsistemas coexisten en un único proceso del sistema operativo, comunicándose a través de múltiples mecanismos coordinados: **canales Go** como `sigChan`, `shutdownChan`, y `completedChan` para coordinación asíncrona de eventos; objetos compartidos incluyendo `Store`, `DiskPieceStore`, y `Manager` con sincronización mediante mutex para acceso seguro; y llamadas directas que permiten invocaciones síncronas entre componentes del mismo proceso como `ov.Announce()` y `mgr.AddPeer()`, optimizando la comunicación intra-proceso.

2.3. Patrones de diseño para optimización de desempeño

La arquitectura de alto rendimiento del sistema se logra mediante la aplicación estratégica de patrones de concurrencia probados que maximizan la eficiencia computacional y de red.

El sistema implementa un **patrón de concurrencia híbrido** que combina múltiples estrategias de paralelización para maximizar el rendimiento y la eficiencia de recursos. La concurrencia se basa en actores ligeros (goroutines) donde cada componente principal ejecuta como una goroutine independiente con responsabilidades específicas, aprovechando que las goroutines requieren solo 2KB de stack inicial comparado con 2MB de hilos tradicionales, mientras el scheduler de Go multiplexa miles de goroutines sobre unos pocos hilos del sistema operativo con cambio de contexto extremadamente rápido medido en nanosegundos versus microsegundos de los hilos convencionales.

La comunicación entre componentes utiliza el patrón Producer-Consumer implementado a través de canales de Go que proporcionan comunicación segura entre goroutines, incluyendo el canal de señales `sigChan` para captura asíncrona de SIGINT/SIGTERM, el canal de completion `completedChan` para notificar finalización de descarga, y el canal de shutdown `shutdownChan` para coordinar cierre limpio de todos los componentes del sistema.

Las operaciones de red aprovechan I/O asíncrono con multiplexing utilizando el runtime de Go que implementa Epoll/Kqueue para multiplexar miles de conexiones TCP simultáneas, non-blocking I/O donde las goroutines se suspenden automáticamente durante operaciones de red liberando recursos, y connection pooling para reutilización eficiente de conexiones TCP especialmente en el protocolo de gossip.

La descarga de piezas se optimiza mediante **paralelización avanzada** que incluye pipeline de requests donde cada conexión peer mantiene una ventana de bloques pendientes, descarga simultánea permitiendo que múltiples peers envíen bloques de la misma pieza concurrentemente, y verificación asíncrona donde el hash SHA-1 se calcula en background sin bloquear el proceso de descarga, maximizando el throughput general del sistema.

Las **optimizaciones de memoria y CPU** incorporan técnicas avanzadas de gestión de recursos que incluyen memory pooling para reutilización de buffers destinados a mensajes de red, zero-copy operations que permiten escritura directa a disco sin copias intermedias en memoria, lazy initialization donde los bitfields remotos se crean únicamente cuando es necesario, y batching donde los announces periódicos agrupan múltiples actualizaciones en un solo mensaje para reducir overhead de red.

Este diseño arquitectural permite que un solo proceso maneje eficientemente cientos de conexiones peer simultáneas y procese miles de mensajes de gossip por segundo, manteniendo consistentemente baja latencia y uso óptimo de recursos del sistema, demostrando la efectividad de combinar concurrencia basada en goroutines con optimizaciones específicas de red y almacenamiento.

3. Comunicación

El sistema distribuido implementa un modelo de comunicación heterogéneo que combina múltiples protocolos especializados, cada uno optimizado para diferentes aspectos de la funcionalidad del sistema. Esta arquitectura multicapa permite separar las responsabilidades de transferencia de datos, descubrimiento de peers, y coordinación del sistema.

3.1. Tipo de comunicación

El sistema implementa una arquitectura de comunicación multicapa con protocolos especializados para diferentes propósitos, donde cada protocolo está optimizado para sus responsabilidades específicas dentro del ecosistema distribuido.

El **protocolo BitTorrent/Peerwire** constituye el núcleo de la transferencia de datos, implementando el protocolo estándar BitTorrent sobre conexiones TCP bidireccionales para la transferencia eficiente de piezas de archivos. El proceso de establecimiento de conexión utiliza un **handshake protocol** de 68 bytes que incluye protocol string length (1 byte siempre 19), protocol string (19 bytes "BitTorrent protocol"), reserved bytes (8 bytes para extensiones del protocolo), InfoHash (20 bytes identificador SHA-1 del torrent), y PeerID (20 bytes identificador único del peer).

Cada mensaje peerwire sigue una **estructura binaria fija** compuesta por length prefix (4 bytes big-endian indicando longitud del mensaje), message ID (1 byte especificando tipo de mensaje 0-9), y payload (longitud variable con datos específicos del mensaje). Los tipos de mensajes implementados incluyen `MsgChoke/MsgUnchoke` para control de flujo de descarga, `MsgInterested/MsgNotInterested` para expresión de interés, `MsgHave` para anuncio de nueva pieza disponible, `MsgBitfield` para estado inicial de piezas disponibles, `MsgRequest` para solicitud de bloque específico, `MsgPiece` para entrega de datos de bloque, `MsgCancel` para cancelación de request pendiente, y `Keep-alive` para mantener conexión viva.

El **protocolo de Overlay Gossip** constituye un sistema de comunicación JSON sobre TCP específicamente diseñado para el mantenimiento de la red superpuesta y el descubrimiento distribuido de peers. Los mensajes del overlay siguen un esquema JSON estandarizado representado por la estructura `wireMsg`, que incluye un campo tipo para identificar la operación (gossip, announce, lookup), el hash del torrent objetivo, una lista de metadatos de proveedores conteniendo dirección, identificador, bytes restantes, y timestamp de última actividad, además de un límite opcional para el número de respuestas esperadas. Esta estructura proporciona comunicación flexible entre nodos sin requerir un protocolo binario rígido.

Los tipos de operaciones soportadas incluyen `announce` para notificar disponibilidad de un torrent, `lookup` para buscar proveedores de un torrent específico, y `gossip` para propagar información de proveedores conocidos. El sistema implementa patrones de comunicación diferenciados: fire-and-forget para announces y gossip que no esperan respuesta, request-response para lookups que esperan lista de proveedores, y timeouts cortos de 1200ms para gossip y 800ms para health checks, optimizando la responsividad de la red.

Para mantener compatibilidad con trackers BitTorrent estándar, el sistema incorpora soporte para **protocolo HTTP/Bencode** como modo fallback. Los announces utilizan HTTP GET con URLs conteniendo parámetros query estándar incluyendo `peer_id`, `port`, `uploaded`, `downloaded`, `left`, y `event`, mientras las scrape requests permiten obtención de estadísticas del torrent como `seeders`, `leechers`, y `completed`. La serialización se maneja mediante bencode encoding/decoding, el estándar binario de BitTorrent para metadatos de torrents y respuestas de tracker.

Como servicio auxiliar, el sistema incluye un **protocolo DNS personalizado** que implementa un sistema DNS distribuido para resolución de nombres en el overlay. Este servicio utiliza UDP DNS resolver en puerto 8053 para consultas de resolución, API REST para registro de registros DNS mediante POST HTTP con JSON, y gossip DNS para sincronización de registros entre servidores DNS distribuidos.

3.2. Comunicación cliente-servidor y servidor-servidor

El sistema arquitectural se fundamenta en un modelo híbrido que combina patrones cliente-servidor con peer-to-peer, maximizando las ventajas de ambos paradigmas según el contexto operacional.

La **comunicación peer-to-peer pura** constituye el núcleo del sistema, donde el protocolo BitTorrent Peerwire implementa comunicación simétrica permitiendo que cada nodo actúe como cliente y servidor simultáneamente mediante conexiones bidireccionales y persistentes. El overlay gossip refuerza esta simetría donde cada nodo mantiene un servidor TCP ejecutando `serveListener` mientras simultáneamente inicia conexiones como

cliente a través de `sendWireMsg`. El descubrimiento distribuido elimina completamente servidores centrales, propagando información epidémicamente entre todos los nodos de manera descentralizada.

Como complemento, la **comunicación cliente-servidor residual** se mantiene para funcionalidades auxiliares y compatibilidad. El tracker HTTP opcional proporciona un modo de compatibilidad donde los nodos actúan como clientes hacia un tracker HTTP centralizado cuando sea necesario, mientras el DNS auxiliar permite que los nodos consulten servidores DNS distribuidos para resolución de nombres de servicios.

La topología de red resultante forma un grafo no estructurado donde cada nodo mantiene conexiones a un subconjunto de peers bootstrap, las conexiones peerwire se establecen dinámicamente basándose en disponibilidad de piezas, el overlay gossip crea una malla parcial para propagación eficiente de información, y health checks periódicos eliminan nodos no responsivos de la topología activa, manteniend connectivity resiliente.

3.3. Comunicación entre procesos

La coordinación intra-proceso se basa en primitivas nativas de Go que implementan el paradigma CSP (Communicating Sequential Processes), proporcionando comunicación eficiente y type-safe entre goroutines concurrentes.

Los **canales de Go** constituyen el mecanismo principal de comunicación inter-goroutine, implementando canales de control mediante `chan struct{}` que incluyen `sigChan` para captura de señales del sistema SIGINT/SIGTERM, `shutdownChan` para coordinación de cierre ordenado de componentes, `completedChan` para notificación de finalización de descarga, y `stopCh` para detener loops de goroutines del overlay de manera controlada. La semántica de canales incluye unbuffered channels para sincronización punto-a-punto bloqueante, buffered channels para desacoplamiento temporal con capacidad limitada, y select statement para multiplexing no bloqueante de múltiples canales simultáneamente.

Los canales de datos facilitan comunicación asíncrona entre producer/consumer goroutines con buffer limitado para prevenir memory leaks durante overload, utilizando select statement para multiplexing eficiente de múltiples canales simultáneamente.

La **shared memory con sincronización** complementa los canales mediante mutex donde `sync.Mutex` y `sync.RWMutex` protegen recursos compartidos críticos: `Store.mu` protege el store distribuido del overlay, `PieceDownload.mutex` sincroniza acceso al estado de descarga, y RWMutex en `DiskPieceStore` permite lecturas concurrentes optimizadas. Los `sync.WaitGroup` proporcionan sincronización para shutdown coordinado entre goroutines y espera de finalización de discovery paralelo, garantizando terminación limpia del sistema.

Las **invocaciones directas** forman el núcleo de la comunicación síncrona entre componentes, donde las llamadas síncronas como `ov.Announce()`, `store.WriteBlock()`, y `mgr.AddPeer()` proporcionan comunicación inmediata para operaciones que requieren confirmación instantánea. El sistema complementa estas llamadas directas con **callbacks** como `OnPieceComplete()` que implementan notificaciones event-driven, permitiendo respuesta asíncrona a eventos del sistema. La arquitectura utiliza **interfaces compartidas** como `PieceStore` y `Manager` que abstraen implementaciones específicas, facilitando el desacoplamiento entre componentes y permitiendo intercambio transparente de implementaciones.

Los **patrones de concurrencia** optimizan el rendimiento mediante estrategias de paralelización probadas que maximizan la utilización de recursos. El sistema implementa

worker pools asignando una goroutine dedicada por conexión peer para procesamiento paralelo, garantizando que cada conexión tenga recursos computacionales dedicados sin interferencia. La arquitectura **producer-consumer** separa la generación de announces y gossip en goroutines especializadas del consumo y procesamiento en otras, optimizando el pipeline de datos. El patrón **fan-out** permite discovery paralelo hacia múltiples peers bootstrap simultáneamente, acelerando la exploración de la red. Los **ticker patterns** coordinan eventos periódicos como gossip cada 8 segundos y health checks cada 10 segundos, manteniendo la sincronización temporal del sistema distribuido.

Las **optimizaciones de comunicación** minimizan la latencia y maximizan el throughput mediante técnicas avanzadas de gestión de conectividad. El **connection pooling** permite reutilización eficiente de conexiones TCP para gossip frecuente, eliminando el overhead de establecimiento de conexiones repetitivo. La técnica de **pipelining** habilita múltiples requests peerwire sin esperar respuestas individuales, maximizando la utilización del ancho de banda disponible. El **batching** agrupa múltiples updates en mensajes de gossip únicos, reduciendo substancialmente el overhead de red. La **lazy propagation** optimiza el gossip enviando únicamente cuando hay cambios significativos en lugar de propagación ciega. El mecanismo de **circuit breaker** implementa health checks que eliminan automáticamente peers no responsivos, evitando timeouts prolongados que degradarían el rendimiento general del sistema.

4. Coordinación

La coordinación en un sistema distribuido sin autoridad central requiere mecanismos sofisticados para sincronizar acciones, gestionar acceso a recursos compartidos, y tomar decisiones colectivas. El sistema BitTorrent implementa múltiples estrategias coordinadas que garantizan operación coherente ante concurrencia masiva y fallos parciales.

4.1. Sincronización de acciones

El sistema implementa múltiples mecanismos de sincronización para coordinar acciones distribuidas tanto a nivel de red como a nivel de proceso, donde la sincronización de información sobre disponibilidad de recursos se logra mediante un **protocolo de anti-entropía distribuido** que utiliza gossip epidémico.

El **gossip periódico** constituye el mecanismo fundamental de sincronización, donde cada 8 segundos cada nodo propaga su estado completo a todos los peers bootstrap mediante la función `periodicGossip`, enviando la lista completa de proveedores por `infoHash` para mantener coherencia distributiva. La función `Store.Merge()` implementa **merge semántico** que logra convergencia automática cuando se reciben actualizaciones conflictivas, aplicando la política Last-Write-Wins basada en timestamps `LastSeen` para resolución determinística de conflictos. Los **health checks distribuidos** operan mediante el proceso `periodicHealthCheck` que cada 10 segundos detecta y elimina peers no responsivos, manteniendo la topología del overlay actualizada y funcional.

La descarga cooperativa de piezas requiere coordinación precisa entre múltiples peers mediante algoritmos de distribución de carga y seguimiento de estado distribuido. El **Round-Robin scheduling** implementado en `DownloadPieceParallel()` distribuye bloques de una pieza entre peers disponibles usando una estrategia circular para balancear la carga equitativamente, donde el sistema calcula el índice del peer usando el módulo del número de bloque entre la cantidad de peers disponibles, asigna el peer seleccionado

al bloque correspondiente, y mantiene el mapeo en la estructura de control de descarga distribuida.

El **estado de descarga distribuida** se rastrea mediante la estructura `PieceDownload` que mantiene el estado global de descarga de cada pieza, incluyendo `blocksPending` para bloques que faltan por descargar, `blocksInProgress` para bloques siendo descargados y el peer asignado, y `blocksReceived` para estadísticas de contribución por peer, proporcionando visibilidad completa del progreso de descarga paralela.

Los eventos críticos del sistema se sincronizan mediante un patrón de coordinación basado en canales que garantiza transiciones de estado consistentes y cierre ordenado del sistema. Los **completion events** utilizan el canal `completedChan` para coordinar la transición crítica de leecher a seeder, activando automáticamente announces de `çompleted"` y broadcasts de disponibilidad que notifican a toda la red sobre la nueva capacidad de servir el archivo completo.

La **shutdown coordination** se implementa mediante el canal `shutdownChan` que ejecuta un protocolo de cierre graceful coordinando múltiples operaciones secuenciales: primero envía announces de `"stopped"` al overlay para notificar la desconexión planificada, luego procede al cierre ordenado de conexiones peer activas, continúa con la detención controlada de procesos de gossip y health checks, y finalmente completa la liberación segura de recursos de almacenamiento, garantizando que el nodo se desconecte limpiamente sin afectar la estabilidad del sistema distribuido.

4.2. Acceso exclusivo a recursos

La gestión de recursos compartidos en un entorno altamente concurrente requiere primitivas de sincronización sofisticadas que balanceen rendimiento con seguridad de acceso.

El sistema gestiona múltiples recursos compartidos mediante una jerarquía de primitivas de sincronización donde el recurso crítico más importante es el **Store del overlay** que mantiene el mapeo `infoHash ->providers`. La protección se implementa mediante `sync.RWMutex` que permite múltiples lectores concurrentes para operaciones como `Lookup` y `ToJSON`, pero garantiza acceso exclusivo para escritores durante `Announce`, `Merge`, y `Replace`, optimizando así el rendimiento de consultas frecuentes mientras mantiene consistencia durante actualizaciones. La **prevención de race conditions** se logra mediante el patrón `defer unlock` que garantiza liberación automática de locks incluso ante panics o returns tempranos, eliminando deadlocks potenciales.

La gestión de conexiones peer requiere sincronización fina para evitar condiciones de carrera durante operaciones críticas de red y transferencia de datos. La **coordinación en Manager de Peers** utiliza `sync.RWMutex` para proteger la lista de peers activos durante operaciones de `AddPeer` y `RemovePeer`, además de broadcasts de mensajes, permitiendo lecturas concurrentes de la lista mientras serializa modificaciones que podrían alterar la topología de conectividad. Complementariamente, un `sync.Mutex` dedicado llamado `downloadsMu` protege específicamente el estado de descargas de piezas, evitando que múltiples goroutines inicien descargas duplicadas del mismo bloque, garantizando uso eficiente de ancho de banda y prevención de trabajo redundante.

La **sincronización de almacenamiento** se gestiona mediante el `DiskPieceStore` que coordina acceso concurrente al sistema de archivos garantizando integridad de datos durante operaciones de lectura y escritura simultáneas. El sistema implementa `sync.RWMutex` dedicado para proteger el bitfield local durante operaciones críticas como escritura de bloques y verificación de completitud de piezas, permitiendo múltiples consultas concurrentes

del estado de completitud mientras serializa actualizaciones que modifican el bitfield. Los **callbacks atómicos** garantizan que las notificaciones `OnPieceComplete` se ejecuten de forma atómica, evitando broadcasts duplicados de mensajes `HAVE` que podrían confundir a peers remotos sobre el estado real de disponibilidad de piezas.

4.3. Toma de decisiones distribuidas

La ausencia de un coordinador central requiere que cada nodo tome decisiones autónomas inteligentes que optimicen tanto su rendimiento individual como la eficiencia global del sistema distribuido.

El sistema implementa varios algoritmos de decisión distribuida sin coordinador central, donde cada nodo toma **decisiones autónomas de selección de peers** basadas en heurísticas sofisticadas. La **piece picker strategy** utiliza el algoritmo `NextPieceFor()` que implementa inicialmente una estrategia "first-needed" optimizada para completar descargas rápidamente, pero puede evolucionar hacia "rarest-first" para optimizar la distribución de piezas raras en la red y mejorar la disponibilidad general del contenido.

Las **heurísticas de selección de peers** priorizan conexiones basadas en criterios múltiples que incluyen disponibilidad de piezas necesarias mediante `RemoteHasPiece()`, estado de choking utilizando `!PeerChoking` para identificar peers dispuestos a enviar datos, latencia y throughput histórico para optimizar rendimiento de red, y timestamp de última actividad `LastSeen` para priorizar peers activos y responsivos.

Sin coordinador central, el sistema resuelve conflictos mediante políticas determinísticas que garantizan convergencia eventual sin requerir consenso explícito. La **resolución de conflictos distribuida** implementa una estrategia Last-Write-Wins (LWW) mediante la función `Merge()` que resuelve conflictos de metadatos de providers comparando timestamps Unix, donde la información más reciente siempre prevalece, eliminando ambigüedad en actualizaciones concurrentes. El sistema incorpora una **ventana de tolerancia** mediante TTL configurado a 90 segundos por defecto, creando un período temporal donde información levemente "stale" se considera válida, reduciendo conflictos causados por relojes desincronizados entre nodos y proporcionando robustez ante variaciones temporales menores.

Aunque no implementa un algoritmo de consenso formal como Raft o PBFT, el sistema logra **consenso implícito mediante gossip** que garantiza consistencia eventual a través de mecanismos probabilísticos robustos. La **convergencia epidémica** se logra mediante gossip periódico que garantiza matemáticamente que la información se propague a toda la red conectada con alta probabilidad, aprovechando las propiedades de propagación epidemiológica para alcanzar todos los nodos accesibles.

La **redundancia y replicación natural** del sistema permite que cada nodo mantenga múltiples copias de la información de providers, tolerando la pérdida de hasta N-1 nodos donde N representa el número total de nodos que conocen un `infoHash` específico, proporcionando alta disponibilidad sin requerir replicación explícita. El mecanismo de **self-healing** permite que el sistema se auto-repare eliminando automáticamente información stale mediante TTL y health checks, manteniendo coherencia distribuida sin requerir intervención manual o coordinación centralizada.

La descarga paralela de datos requiere **decisiones distribuidas de scheduling** que optimicen el uso de recursos sin coordinación centralizada, donde cada nodo toma decisiones autónomas de distribución de carga. El **Round-Robin distribuido** permite que cada nodo ejecute independientemente un algoritmo de distribución circular que asigna

bloques de piezas a peers usando el módulo del número de bloque entre la cantidad de peers disponibles, garantizando distribución equitativa de trabajo sin requerir coordinación central y evitando tanto la sobrecarga de peers individuales como la subutilización de recursos de red disponibles.

El **retry automático** maneja fallos de peers mediante un mecanismo de recuperación que se activa cuando un peer se desconecta, identificando bloques en progreso asignados al peer caído, marcándolos como pendientes, y redistribuyéndolos entre peers alternativos usando el mismo algoritmo Round-Robin, manteniendo progreso de descarga sin intervención manual y evitando bloqueos indefinidos. La **adaptación dinámica** permite que las decisiones de scheduling se ajusten en tiempo real al estado cambiante de la red, incluyendo peers que se unen o salen dinámicamente y cambios en disponibilidad de piezas, maximizando la eficiencia de transferencia ante condiciones variables de red.

5. Nombrado y Localización

La identificación y localización de recursos en un sistema distribuido requiere esquemas de nombrado únicos y algoritmos de descubrimiento eficientes que operen sin depender de autoridades centrales. El sistema implementa un modelo híbrido que combina identificadores criptográficos para garantizar unicidad global con mecanismos de descubrimiento distribuido para localización dinámica.

5.1. Identificación de los datos y servicios

El sistema implementa un esquema de identificación jerárquico que combina múltiples espacios de nombres, donde los datos se identifican mediante un sistema criptográfico basado en hashes que garantiza unicidad global y los servicios utilizan identificadores multicapa para diferentes niveles de abstracción.

La **identificación de recursos y contenido** utiliza un sistema criptográfico robusto donde el **InfoHash primario** constituye un identificador único de 160 bits (20 bytes) generado mediante SHA-1 del diccionario ‘info‘ del archivo ‘.torrent‘, garantizando unicidad matemática global para cada recurso distribuido independientemente de su localización física o lógica.

La **codificación para transporte** adapta el `infoHash` según el protocolo de comunicación específico, utilizando representación hexadecimal mediante `fmt.Sprintf("%x", infoHash[:])` para logs y debugging que facilite inspección manual, codificación URL-encoded a través de `percentEncode20(infoHash)` para HTTP announces compatible con estándares web, formato binario directo de 20 bytes para handshakes peerwire optimizando eficiencia de red, y representación string mediante `InfoHashEncoded` para uso como keys en mapas del overlay.

Los **metadatos del archivo** complementan la identificación criptográfica con información contextual extraída del torrent, incluyendo `FileName` para el nombre del archivo a descargar, `FileLength` especificando el tamaño total en bytes, `PieceLength` definiendo el tamaño de cada pieza típicamente entre 256KB y 1MB, y `ExpectedHashes` conteniendo el array de hashes SHA-1 de cada pieza para verificación rigurosa de integridad durante transferencia.

Los nodos del sistema se identifican mediante un **esquema de identificación multicapa** que combina identificadores únicos, información de conectividad, y metadatos de

estado. El **PeerID único** constituye un identificador de 20 bytes generado usando entropy criptográfica donde el algoritmo genera 6 bytes aleatorios seguros, los codifica en hexadecimal, y los concatena con el prefijo del cliente `-JC0001-` para formar un identificador único de formato estándar BitTorrent con formato `-JC0001-` seguido de 12 caracteres hexadecimales aleatorios, garantizando unicidad estadística entre miles de millones de nodos distribuidos.

La **dirección de red** proporciona identificación de conectividad TCP/IP mediante múltiples componentes coordinados: `hostname` especificando la dirección IP o nombre DNS del nodo, `port` definiendo el puerto TCP para conexiones peerwire, `overlay-port` estableciendo el puerto TCP para protocolo de gossip con valor por defecto 6000, y `Addr` representando la combinación `ip:puerto` usada como key única para indexación eficiente en estructuras de datos distribuidas.

Los **metadatos de estado** complementan la identificación con información contextual del nodo incluyendo `Left` indicando bytes restantes por descargar donde 0 representa un seeder completo, `LastSeen` almacenando el timestamp Unix de última actividad para algoritmos de anti-entropía, y `TTL` definiendo el tiempo de vida de la información con valor por defecto de 90 segundos para garbage collection automático de información obsoleta.

El sistema DNS distribuido implementa su propio espacio de nombres mediante **servicios auxiliares de identificación** que proporcionan resolución descentralizada de nombres. Los **registros DNS** implementan mapeo 'nombre -¿IP' con metadatos temporales que incluyen timestamps de creación y TTL para expiración automática, mientras la **resolución jerárquica** permite que el sistema resuelva nombres mediante el DNS distribuido y utilice esas direcciones resueltas para bootstrap del overlay, proporcionando un puente entre nombrado humano-legible y identificación de red.

5.2. Ubicación y Localización de los datos y servicios

La localización efectiva de recursos en un entorno distribuido requiere algoritmos que balanceen eficiencia de búsqueda con overhead de red, operando sin coordinador central mediante técnicas probabilísticas y caché distribuido.

El sistema implementa múltiples algoritmos de localización distribuida que operan sin coordinador central, donde la **localización básica mediante gossip epidémico** utiliza propagación epidemiológica de información de proveedores para descubrimiento eficiente. El **anuncio distribuido** implementado mediante `Announce` permite que un nodo que posee un recurso se registre localmente y propague su disponibilidad a través de la red overlay, estableciendo presencia global sin requerir coordinación centralizada.

La **búsqueda híbrida** ejecutada por `Lookup` combina información local con consultas distribuidas mediante un algoritmo de cuatro etapas: primero realiza consulta local al `Store` con complejidad O(1) aplicando filtrado por TTL para información fresca, luego si los resultados son insuficientes consulta hasta 3 peers bootstrap con fanout limitado para evitar saturación de red, continúa con merge inteligente de resultados ordenados por `LastSeen` descendente priorizando información reciente, y finalmente aplica el límite de respuestas solicitado para controlar el volumen de datos transferidos.

Para bootstrapping inicial o recuperación ante particiones de red, el sistema implementa un **discovery avanzado con BFS distribuido** que proporciona exploración exhaustiva de la topología cuando los mecanismos básicos son insuficientes. La **búsqueda en anchura distribuida** utiliza el método `Discover` que implementa BFS con TTL para explorar sistemáticamente la topología de red, garantizando cobertura completa den-

tro de la profundidad máxima especificada mientras evita explosiones exponenciales de consultas.

La **paralelización con sincronización** optimiza el rendimiento ejecutando consultas a peers concurrentemente usando goroutines con `sync.WaitGroup` para coordinación precisa, maximizando el paralelismo disponible sin crear condiciones de carrera. Los mecanismos de **prevención de loops** utilizan el mapa `seen` para evitar revisitar nodos ya explorados mientras el TTL limita la profundidad de búsqueda, previniendo tanto ciclos infinitos como consumo excesivo de recursos de red.

El sistema combina **estrategias de localización reactiva y proactiva** adaptadas al contexto operacional para optimizar tanto latencia de búsqueda como overhead de mantenimiento. La **localización reactiva bajo demanda** activa `Lookup` únicamente cuando se necesita descargar un torrent específico, ejecuta consultas `queryPeerLookup` dirigidas a peers específicos, y utiliza timeouts cortos de 800ms para evitar latencia alta que degrade la experiencia de usuario, minimizando el tráfico de red mediante búsquedas puntuales.

La **localización proactiva de mantenimiento** opera continuamente mediante gossip periódico cada 8 segundos que propaga disponibilidad de recursos conocidos, health checks cada 10 segundos que validan conectividad de peers conocidos, y discovery inicial durante bootstrap del overlay que establece conectividad fundamental, manteniendo la red overlay actualizada sin esperar consultas específicas.

El enfoque de **localización adaptativa** ajusta parámetros dinámicamente mediante TTL que adapta la vida útil de información según el churn observado de la red, fanout limitado a 3 peers que balancea cobertura de búsqueda versus overhead de comunicación, y ordenamiento por `LastSeen` que prioriza información fresca sobre datos potencialmente obsoletos, optimizando la eficiencia global del sistema.

Para soporte de infraestructura, el sistema incluye **resolución DNS distribuida** que proporciona servicios de nombrado descentralizado mediante servidor DNS over UDP en puerto 8053 para resolución eficiente de consultas tipo A, API REST para registro que ofrece endpoint HTTP para agregar registros DNS dinámicamente, gossip entre DNS servers que sincroniza registros entre nodos DNS distribuidos, e integración con overlay que permite usar nombres resueltos como direcciones bootstrap, creando un puente entre nombrado humano-legible y descubrimiento automatizado.

Los algoritmos de localización están diseñados para operar con **tolerancia a fallos integral** ante fallos parciales de red o nodos. La **redundancia de información** garantiza que múltiples nodos mantengan copias de la información de proveedores, eliminando puntos únicos de fallo. Los **timeouts agresivos** de 800ms-1200ms permiten detección rápida de nodos no responsivos sin esperas prolongadas que degraden el rendimiento.

Las **estrategias de fallback** permiten usar tracker HTTP como respaldo cuando el overlay falla completamente, manteniendo funcionalidad básica durante interrupciones. Los mecanismos de **self-healing** combinan gossip periódico, TTL automático, y health checks para eliminar automáticamente información stale sin intervención manual. El **bootstrapping robusto** utiliza discovery con múltiples seeds iniciales y BFS para aumentar drásticamente la probabilidad de éxito de conexión inicial, garantizando operación confiable incluso ante condiciones adversas de red.

6. Consistencia y Replicación

Esta sección describe el modelo de consistencia y las estrategias de replicación usadas por el overlay para mantener disponibilidad y convergencia en presencia de churn y fallos.

6.1. Modelo de Datos y Estructura de las Réplicas

El sistema mantiene réplicas distribuidas de la información crítica necesaria para localizar proveedores y coordinar la propagación de metadatos. En particular, el mapeo `infoHash ->ProviderMeta` contiene, para cada proveedor, la dirección de red, los bytes pendientes por descargar y un timestamp de última actividad representado por `LastSeen`. Además, cada nodo conserva metadatos de conectividad (listas de peers y rutas de gossip) que permiten la propagación y el health checking periódico.

Cada nodo expone un `Store` local que actúa como réplica parcial del espacio de nombres de proveedores. El particionado es natural: un nodo almacena información de los torrents que conoce o en los que participa, y la redundancia es adaptativa según la participación observada en el overlay. Para evitar acumulación de entradas obsoletas, los registros cuyo `LastSeen` excede el valor de `TTL` (por defecto 90 segundos) se descartan automáticamente durante las operaciones de consulta.

6.2. Estrategias de Replicación

La replicación principal es epidémica: un proceso periódico serializa y transmite la información de proveedores hacia peers vecinos de forma distribuida. Un ticker periódico (cada 8 segundos) activa la rutina de gossip que itera sobre los `infoHash` conocidos y empuja actualizaciones paralelizadas mediante goroutines, aprovechando la redundancia temporal del protocolo para tolerar pérdidas puntuales de mensajes. En la práctica se usa una política de propagación de baja latencia (conexiones TCP efímeras y timeouts de 1200ms) y un esquema push-based que prioriza throughput y disponibilidad; la repetición periódica del gossip compensa la ausencia de confirmaciones individuales.

Para registros nuevos, el sistema combina almacenamiento local inmediato con propagación rápida vía gossip. El factor de réplicas se ajusta según el tamaño del overlay: en despliegues pequeños (menos de 10 nodos) se puede mantener replicación completa, mientras que en overlays grandes se controla el fanout (por ejemplo, fanout = 3) para limitar overhead. Adicionalmente, las copias en nodos bootstrap aumentan la probabilidad de disponibilidad durante fallos.

6.3. Modelo de Consistencia

El sistema aplica consistencia eventual con resolución determinista mediante `Last-Write-Wins`. En los procedimientos de merge, se compara el campo `LastSeen` entre entradas conflictivas y se conserva aquella con timestamp más reciente. Esta estrategia evita la coordinación fuerte entre nodos y garantiza convergencia eventual aun cuando múltiples actualizaciones concurrentes ocurran.

La replicación epidémica asegura convergencia: en ausencia de actualizaciones, las réplicas alcanzan el mismo estado gracias al anti-entropía periódico. Durante churn puede aparecer inconsistencia temporal en vistas locales, la cual se corrige con el tiempo. El campo `LastSeen` proporciona monotonía temporal y permite idempotencia en los merges; conflictos se resuelven por timestamp, entradas duplicadas por dirección se consolidan

manteniendo una sola entrada por `Addr`, y la información stale se filtra usando el umbral de `TTL`. Los peers muertos se detectan por health checks periódicos (cada 10 segundos) y timeouts de conexión configurables (por ejemplo 20 segundos), lo que permite limpiar rápidamente entradas inválidas.

6.4. Garantías de Confiabilidad

El sistema opera con un modelo de "soft-state": los datos se mantienen en memoria para favorecer rendimiento y disponibilidad y no se persisten en disco por defecto. Un nodo que reinicia puede reconstruir su vista conectándose a peers activos y recuperando estado vía gossip. Esta aproximación elimina la complejidad de persistencia distribuida mientras mantiene alta disponibilidad mediante replicación en memoria.

La limpieza y recuperación se basan en mecanismos combinados de self-healing que operan automáticamente sin intervención manual. La recolección temporal mediante `TTL` elimina información obsoleta, el health checking proactivo valida conectividad de peers periódicamente, y la anti-entropía mediante gossip periódica corrige inconsistencias graduales. Los timeouts agresivos (800ms–1200ms en conexiones) contribuyen a una detección rápida de fallos y a mantener el conjunto de réplicas coherente, asegurando que el sistema se auto-repare ante condiciones adversas.

7. Tolerancia a Fallas

El sistema implementa múltiples capas de tolerancia a fallos para mantener disponibilidad y confiabilidad ante diversos tipos de interrupciones y condiciones adversas de red.

7.1. Taxonomía de Fallos y Estrategias de Detección

El sistema BitTorrent distribuido maneja múltiples tipos de fallos de forma diferenciada según su naturaleza e impacto. Los **fallos de nodos** (crash failures) incluyen desconexión abrupta de peers durante descarga o participación en overlay, afectando tanto la transferencia de datos como la propagación de metadatos. Los **fallos de comunicación** abarcan timeouts de red, pérdida de paquetes y particiones temporales que pueden aislar subconjuntos de nodos del resto de la red. Los **fallos de componentes** afectan servicios específicos como el tracker HTTP, nodos DNS distribuido o servicios auxiliares, requiriendo mecanismos de fallback para mantener operación. Adicionalmente, el sistema debe manejar **fallos bizantinos limitados** donde peers maliciosos envían datos corruptos o metadatos incorrectos, pero siguen parcialmente el protocolo.

La detección de fallos opera en múltiples capas con diferentes granularidades temporales para identificar rápidamente interrupciones y activar mecanismos de recuperación. La **detección a nivel de conexión TCP** utiliza timeouts agresivos para conexiones individuales, permitiendo identificación rápida de peers no responsivos durante handshakes y transferencias de datos. El **health checking periódico del overlay** implementa verificación proactiva cada 10 segundos que valida conectividad de peers conocidos y actualiza el estado de disponibilidad en el `Store` distribuido. La **verificación de conectividad durante discovery** realiza tests de conectividad antes de explorar nodos mediante `BFS`, evitando propagación de información sobre peers inaccesibles.

7.2. Recuperación ante Fallos de Nodos

El sistema implementa recuperación automática ante desconexión de peers durante la transferencia de datos mediante algoritmos que detectan peers inactivos, marcan bloques en progreso como pendientes y los redistribuyen entre peers alternativos. Cuando un peer se desconecta durante una transferencia activa, el sistema identifica todos los bloques asignados a ese peer, los marca como disponibles para reasignación, y aplica el algoritmo de distribución `round-robin` para balancear la carga entre peers restantes.

La reasignación de bloques fallidos se distribuye entre peers disponibles usando `round-robin` para evitar sobrecarga de peers individuales y mantener throughput óptimo. El algoritmo calcula el índice de peer objetivo usando `blockIndex % len(availablePeers)` y asigna el bloque al peer correspondiente, garantizando distribución equitativa de trabajo de recuperación sin requerir coordinación centralizada.

7.3. Tolerancia a Particiones de Red

El diseño distribuido permite que subconjuntos del sistema continúen operando durante particiones de red mediante arquitectura descentralizada que no depende de coordinadores centrales. Los **overlays aislados** permiten que los nodos en cada partición mantengan información local y continúen sirviendo consultas basándose en su conocimiento cached, proporcionando disponibilidad parcial durante interrupciones de conectividad. La **degradación gradual** asegura que la información se vuelva progresivamente stale según los valores de `TTL` configurados, pero no se pierde abruptamente, permitiendo operación continuada con datos levemente obsoletos. La **reconexión automática** activa mecanismos de gossip que resincronizan automáticamente el estado cuando se restaura la conectividad, convergiendo hacia consistencia global sin intervención manual.

El algoritmo de discovery robusto implementa búsqueda en anchura (`BFS`) distribuida con limitación `TTL` para prevenir explosiones exponenciales de consultas durante particiones. El proceso comienza con nodos bootstrap especificados, verifica conectividad usando timeouts de 800ms para detección rápida de fallos, ejecuta consultas `lookup` en paralelo mediante `goroutines` para maximizar throughput, y expande la cola con nuevos nodos descubiertos hasta alcanzar la profundidad TTL configurada. La estrategia utiliza mapas de nodos visitados (`seen`) para prevenir loops y redundancia, maximizando la probabilidad de éxito incluso con fallos parciales de nodos bootstrap o conectividad intermitente.

7.4. Mecanismos de Fallback y Degradación

Cuando el overlay falla completamente, el sistema activa un mecanismo de fallback que mantiene funcionalidad básica mediante infraestructura centralizada tradicional. El proceso construye URLs de announce HTTP con parámetros estándar BitTorrent incluyendo `info_hash`, `peer_id`, `port`, y estadísticas de descarga, envía requests GET con timeout de 10 segundos para evitar bloqueos prolongados, y decodifica respuestas `bencode` del tracker para extraer listas de peers. El sistema verifica campos de `failure reason` para detección de errores y extrae listas de peers para modo de operación degradado, permitiendo continuar descargas cuando la infraestructura distribuida no está disponible.

El sistema implementa degradación escalonada que adapta funcionalidad según disponibilidad de recursos y condiciones de red. La transición **Overlay → Tracker HTTP** se activa cuando el overlay no encuentra peers suficientes, consultando el tracker centralizado como respaldo. La degradación **DNS distribuido → DNS público** permite usar

resolución estándar cuando el servicio DNS propio falla, manteniendo conectividad básica. En redes con alta latencia, el sistema puede degradar de **Gossip → Polling** mediante consultas puntuales menos frecuentes que reduzcan overhead de red. Cuando hay pocos peers disponibles, la **Descarga paralela → Secuencial** reduce paralelismo para evitar competencia excesiva por recursos limitados y optimizar throughput global.

7.5. Garantías de Disponibilidad

El sistema proporciona redundancia en múltiples capas para eliminar puntos únicos de fallo y garantizar disponibilidad continua. La **redundancia de datos** se logra mediante replicación natural donde los bloques se distribuyen entre múltiples seeders y leechers, eliminando dependencia de nodos específicos para acceso a contenido. La **redundancia de metadata** replica información de proveedores vía gossip epidémico, asegurando que múltiples nodos mantengan conocimiento sobre disponibilidad de recursos. La **redundancia de servicios** implementa múltiples nodos DNS distribuido y múltiples puntos de entrada al overlay, evitando interrupciones por fallos de servicios auxiliares. La **redundancia de rutas** permite que el discovery `BFS` encuentre múltiples caminos hacia recursos, proporcionando alternativas cuando rutas específicas fallan.

Las garantías específicas del sistema proporcionan objetivos cuantificables de disponibilidad y recuperación. El `MTTR` (Mean Time To Recovery) objetivo es menor a 20 segundos para detección de peer muerto mediante health checking periódico. El `Block recovery time` se mantiene bajo 5 segundos para reasignar bloques de peer desconectado usando algoritmos de balanceeo eficientes. El `Discovery success rate` supera 95 % cuando al menos 2 nodos bootstrap permanecen activos, garantizando conectividad inicial robusta. La `Partition tolerance` permite operación continua con más de 30 % de nodos conectados, manteniendo funcionalidad parcial durante interrupciones mayores. La `Gossip convergence` alcanza sincronización en menos de 30 segundos para cambios en overlay estable.

A pesar de la robustez del diseño, existen algunos puntos de fallo únicos que representan limitaciones inherentes del sistema. La **dependencia de bootstrap** requiere al menos un nodo bootstrap responsivo para discovery inicial, creando una dependencia crítica durante arranque. La **disponibilidad de contenido** se ve comprometida si todos los seeders de un torrent específico fallan simultáneamente, haciendo el contenido temporalmente indisponible. La **fragmentación de overlay** puede ocurrir durante particiones prolongadas, llevando a overlays desincronizados que requieren reconexión manual. La **exhaustión de recursos** puede saturar nodos con conectividad limitada durante cargas pico, degradando rendimiento general del sistema.

El `Discovery success rate` supera 95 % cuando al menos 2 nodos bootstrap permanecen activos, garantizando conectividad inicial robusta. La `Partition tolerance` permite operación continua con más de 30 % de nodos conectados, manteniendo funcionalidad parcial durante interrupciones mayores. La `Gossip convergence` alcanza sincronización en menos de 30 segundos para cambios en overlay estable.

7.6. Limitaciones y Puntos de Fallo

A pesar de la robustez del diseño, existen algunos puntos de fallo únicos que representan limitaciones inherentes del sistema. La **dependencia de bootstrap** requiere al menos un nodo bootstrap responsivo para discovery inicial, creando una dependencia

crítica durante arranque. La **disponibilidad de contenido** se ve comprometida si todos los seeders de un torrent específico fallan simultáneamente, haciendo el contenido temporalmente indisponible.

La **fragmentación de overlay** puede ocurrir durante particiones prolongadas, llevando a overlays desincronizados que requieren reconexión manual. La **exhaustión de recursos** puede saturar nodos con conectividad limitada durante cargas pico, degradando rendimiento general del sistema.

8. Seguridad

Esta sección analiza las amenazas de seguridad, vulnerabilidades y mecanismos de protección implementados en el sistema BitTorrent distribuido.

8.1. Modelo de Amenazas y Superficie de Ataque

El sistema BitTorrent distribuido enfrenta múltiples vectores de ataque debido a su naturaleza descentralizada y la ausencia de autoridades centrales de confianza. Las **amenazas de integridad** incluyen corrupción de datos durante transferencia y manipulación de metadata del overlay que puede comprometer la exactitud de la información de proveedores. Las **amenazas de disponibilidad** abarcan ataques `DoS`, **eclipse attacks** que aíslan nodos legítimos, y **resource exhaustion** que saturan recursos de sistema. Las **amenazas de confidencialidad** permiten espionaje de tráfico y análisis de patrones de descarga que pueden revelar hábitos de usuarios.

Las **amenazas de identidad** facilitan suplantación de peers y falsificación de `announcements`, mientras que las **amenazas bizantinas** involucran nodos maliciosos que siguen parcialmente el protocolo para causar daño sutil y difícil de detectar. Los puntos de entrada para ataques incluyen el protocolo de `handshake` durante verificación de `InfoHash` y `PeerID`, el protocolo de `gossip` con mensajes JSON no autenticados entre nodos del overlay, la transferencia de bloques con datos de payload y metadatos de piezas, el mecanismo de discovery mediante `BFS` distribuido, y servicios auxiliares como DNS distribuido y tracker HTTP de fallback.

8.2. Mecanismos de Integridad y Autenticación

El sistema implementa verificación de integridad usando `SHA-1` para cada pieza descargada, comparando el hash calculado contra el hash esperado del archivo torrent para detectar corrupción accidental o maliciosa. El protocolo `peerwire` incluye verificación mutua de `InfoHash` durante el handshake, donde cada peer valida que el `InfoHash` objetivo coincida exactamente con el torrent deseado, rechazando inmediatamente conexiones con `InfoHash` incorrecto para prevenir participación accidental en swarms equivocados.

Los `PeerID` se generan usando entropy criptográfica mediante la generación de 6 bytes aleatorios seguros que se codifican en hexadecimal y se concatenan con el prefijo del cliente `-JC0001-`, proporcionando identificación única estadística entre miles de millones de nodos distribuidos. Sin embargo, esta generación no incluye prueba de trabajo computacional, lo que permite la creación rápida de múltiples identidades falsas para ataques `Sybil`.

8.3. Vulnerabilidades de Comunicación

La implementación actual transmite todos los datos en texto plano, exponiendo información sensible a interceptación pasiva y activa. El **tráfico** `peerwire` incluye handshakes, mensajes de control y datos de payload completamente no cifrados, permitiendo que atacantes observen directamente qué contenido está siendo transferido. El **gossip del overlay** transmite mensajes JSON con metadatos críticos como identificadores de contenido, direcciones IP y puertos de peers, timestamps de actividad y límites de consulta sin cifrado alguno, exponiendo completamente la topología del overlay y patrones de participación a cualquier atacante que intercepre el tráfico.

El **DNS distribuido** opera mediante consultas y respuestas UDP sin autenticación, mientras que el **tracker HTTP** utiliza announces y scrapes sobre HTTP plano. Un atacante pasivo puede obtener información sensible observando patrones de comunicación mediante **content fingerprinting** donde `InfoHashes` identifican únicamente contenido específico, **peer profiling** mediante análisis de IPs y puertos que revela participación en swarms, **timing attacks** usando patrones temporales de gossip para revelar topología del overlay, y **traffic volume analysis** donde el volumen de datos transferidos indica tipo y popularidad de contenido.

8.4. Ataques contra el Overlay Distribuido

Un atacante puede aislar nodos legítimos controlando sus conexiones mediante `eclipse attacks` que incluyen **bootstrap poisoning** para controlar nodos bootstrap y dirigir discovery hacia nodos maliciosos, **gossip manipulation** que propaga selectivamente información falsa para aislar víctimas, y `DHT` pollution mediante anuncio de metadata falsa para peers específicos en el contexto del overlay.

La ausencia de autenticación fuerte permite `Sybil attacks` donde un atacante crea múltiples identidades falsas rápidamente debido a que la generación de `PeerID` no requiere esfuerzo computacional significativo. Los ataques de **resource exhaustion** están diseñados para agotar recursos de nodos legítimos mediante **connection flooding** que abre múltiples conexiones TCP sin completar handshake, **gossip amplification** enviando mensajes gossip de gran tamaño para saturar ancho de banda, **discovery bombing** que activa `BFS` discovery con `TTL` alto para causar explosión exponencial de consultas, y **piece request storms** solicitando masivamente bloques inexistentes para saturar I/O de disco.

8.5. Mitigaciones Implementadas

El sistema incluye mecanismos básicos para limitar el impacto de ataques mediante timeouts agresivos y rate limiting. Los **connection timeouts** de 800ms a 5 segundos limitan el tiempo de conexiones colgantes, mientras que el `TTL` en discovery establece límites de profundidad `BFS` para prevenir explosión exponencial de consultas. El **fanout limitado** con máximo 3 peers por `lookup` reduce amplificación de tráfico, y el **health checking** elimina automáticamente peers no responsivos cada 10-20 segundos.

La verificación de integridad opera en múltiples niveles para detectar ataques sofisticados. A **nivel de protocolo**, durante el handshake BitTorrent cada peer envía un paquete de 68 bytes conteniendo identificador de protocolo, flags reservadas, `InfoHash` objetivo y `PeerID` propio, donde el receptor valida que el protocolo sea correcto y que el `InfoHash` coincida exactamente, rechazando inmediatamente conexiones incorrectas. A **nivel de contenido**, cada pieza completa se verifica calculando su hash `SHA-1` y

comparándolo contra el hash esperado del archivo torrent, rechazando piezas con discrepancias y marcándolas para re-descarga. A **nivel de aplicación**, se valida la estructura de mensajes JSON en el overlay.

El diseño resiliente incorpora **descentralización** que elimina puntos únicos de fallo (`SPOF`) reduciendo la superficie de ataque crítico, **redundancia** mediante múltiples paths y réplicas que limitan el impacto de nodos comprometidos, y capacidades de **self-healing** donde el gossip epidémico y `TTL` automáticamente purifican información corrupta.